

This paper is a pre-print of the paper *The Power of Non-Determinism in Higher-Order Implicit Complexity* which has been accepted for publication at the *European Symposium for Programming* (ESOP 2017).

The text is (almost) identical to the published version, but the present work includes an appendix containing full proofs of all the results in the paper.

The Power of Non-Determinism in Higher-Order Implicit Complexity ^{*}

Characterising Complexity Classes using Non-deterministic Cons-free Programming

Cynthia Kop and Jakob Grue Simonsen

Department of Computer Science, University of Copenhagen (DIKU)
kop@di.ku.dk simonsen@di.ku.dk

Abstract. We investigate the power of non-determinism in purely functional programming languages with higher-order types. Specifically, we consider *cons-free* programs of varying data orders, equipped with explicit non-deterministic choice. Cons-freeness roughly means that data constructors cannot occur in function bodies and all manipulation of storage space thus has to happen indirectly using the call stack.

While cons-free programs have previously been used by several authors to characterise complexity classes, the work on *non-deterministic* programs has almost exclusively considered programs of data order 0. Previous work has shown that adding explicit non-determinism to cons-free programs taking data of order 0 does not increase expressivity; we prove that this—dramatically—is not the case for higher data orders: adding non-determinism to programs with data order at least 1 allows for a characterisation of the entire class of elementary-time decidable sets.

Finally we show how, even with non-deterministic choice, the original hierarchy of characterisations is restored by imposing different restrictions.

Keywords: implicit computational complexity, cons-free programming, EXPTIME hierarchy, non-deterministic programming, unitary variables

1 Introduction

Implicit complexity is, roughly, the study of how to create bespoke programming languages that allow the programmer to write programs which are guaranteed to (a) *only* solve problems within a certain complexity class (e.g., the class of polynomial-time decidable sets of binary strings), and (b) to be able to solve *all* problems in this class. When equipped with an efficient execution engine, the programs of such a language may themselves be guaranteed to run within the complexity bounds of the class (e.g., run in polynomial time), and the plethora of means available for analysing programs devised by the programming language

^{*} The authors are supported by the Marie Skłodowska-Curie action “HORIP”, program H2020-MSCA-IF-2014, 658162 and by the Danish Council for Independent Research Sapere Aude grant “Complexity via Logic and Algebra” (COLA).

community means that methods from outside traditional complexity theory can conceivably be brought to bear on open problems in computational complexity.

One successful approach to implicit complexity is to syntactically constrain the programmer’s ability to create new data structures. In the seminal paper [12], Jones introduces *cons-free programming*. Working with a small functional programming language, cons-free programs are *read-only*: recursive data cannot be created or altered (beyond taking sub-expressions), only read from input. By imposing further restrictions on *data order* (i.e., order 0 = integers, strings; order 1 = functions on data of order 0; etc.) and recursion scheme (e.g., full/tail/primitive recursion), classes of cons-free programs turn out to characterise various deterministic classes in the time and space hierarchies of computational complexity.

However, Jones’ language is deterministic and, perhaps as a result, his characterisations concern only deterministic complexity classes. It is tantalising to consider the method in a non-deterministic setting: could adding non-deterministic choice to Jones’ language increase its expressivity; for example, from P to NP?

The immediate answer is *no*: following Bonfante [4], adding a non-deterministic choice operator to cons-free programs with data order 0 makes no difference in expressivity—deterministic or not, they characterise P. However, the details are subtle and depend on other features of the language; when only primitive recursion is allowed, non-determinism *does* increase expressivity from L to NL [4].

While many authors consider the expressivity of higher types, the interplay of higher types and non-determinism is not fully understood. Jones obtains several hierarchies of deterministic complexity classes by increasing data orders [12], but these hierarchies have at most an exponential increase between levels. Given the expressivity added by non-determinism, it is *a priori* not evident that similarly “tame” hierarchies would arise in the non-deterministic setting.

The purpose of the present paper is to investigate the power of *higher-order* (cons-free) programming to characterise complexity classes. The main surprise is that while non-determinism does not add expressivity for first-order programs, the combination of second-order (or higher) programs and non-determinism characterises the full class of elementary-time decidable sets—and increasing the order beyond second-order programs does not further increase expressivity. However, we will also show that there are simple changes to the restrictions that allow us to obtain a hierarchy of characterisations as in the deterministic setting.

Proofs for the results in this paper are all available in the appendix.

1.1 Overview and contributions

We define a purely functional programming language with non-deterministic choice and, following Jones [12], consider the restriction to *cons-free* programs.

Our results are summarised in Figure 1. For completeness, we have also included the results from [12]; although the language used there is slightly more syntactically restrictive than ours, the results easily generalise provided we limit interest to *deterministic* programs, where the **choose** operator is not used. As the technical machinations involved to procure the results for a language with

	data order 0	data order 1	data order 2	data order 3
cons-free deterministic	$P = EXP^0TIME$	$EXP = EXP^1TIME$	EXP^2TIME	EXP^3TIME
cons-free tail-recursive deterministic	$L = EXP^{-1}SPACE$	$PSPACE = EXP^0SPACE$	EXP^1SPACE	EXP^2SPACE
cons-free primitive recursive deterministic	$L = EXP^{-1}SPACE$	$P = EXP^0TIME$	$PSPACE = EXP^0SPACE$	$EXP = EXP^1TIME$

The characterisations obtained in [12], transposed to the more permissive language used here. This list (and the one below) should be imagined as extending infinitely to the right. The “limit” for all rows (i.e., all finite data orders allowed) characterises **ELEMENTARY**, the class of elementary-time decidable sets.

	data order 0	data order 1	data order 2	data order 3
cons-free	P	ELEMENTARY	ELEMENTARY	ELEMENTARY
cons-free unitary variables	$P = EXP^0TIME$	$EXP = EXP^1TIME$	EXP^2TIME	EXP^3TIME

The characterisations obtained by allowing non-deterministic choice. As above, the “limit” where all data orders are allowed characterises **ELEMENTARY** (for both rows).

	arrow depth 0	arrow depth 1	arrow depth 2	arrow depth 3
cons-free	P	ELEMENTARY	ELEMENTARY	ELEMENTARY

The characterisations obtained by allowing non-deterministic choice and considering arrow depth as the variable factor rather than data order

Fig. 1. Overview of the results discussed or obtained in this paper.

full recursion are already intricate and lengthy, we have not yet considered the restriction to tail- or primitive recursion in the non-deterministic setting.

Essentially, our paper has two major contributions: (a) we show that previous observations about the increase in expressiveness when adding non-determinism change dramatically at higher types, and (b) we provide two characterisations of the **EXPTIME** hierarchy using a non-deterministic language—which may provide a basis for future characterisation of common non-deterministic classes as well.

Note that (a) is highly surprising: As evidenced by early work of Cook [6] merely adding full non-determinism to a restricted (i.e., non-Turing complete) computation model may result in it still characterising a *deterministic* class of problems. This also holds true for cons-free programs with non-determinism, as shown in different settings by Bonfante [4], by de Carvalho and Simonsen [7], and by Kop and Simonsen [14], all resulting only in characterisations of deterministic classes such as P . With the exception of [14], all of the above attempts at adding non-determinism consider data order at most 0, and one would expect few changes when passing to higher data orders. This turns out to be patently false as simply increasing to data order 1 already results in an explosion of expressive power.

1.2 Overview of the ideas in the paper

Cons-free programs (Definition 5) are, roughly, functional programs where function bodies are allowed to contain constant data and substructures of the function arguments, but *no data constructors*—e.g., clauses $\text{tl } (x::xs) = xs$ and $\text{tl } [] = []$ are both allowed, but $\text{append } (x::xs) \text{ } ys = x::(\text{append } xs \text{ } ys)$ is not.¹ This restriction severely limits expressivity, as it means no new data can be created.

A key idea in Jones’ original work on cons-free programming is *counting*: expressions which represent numbers and functions to calculate with them. It is not in general possible to represent numbers in the usual unary way as 0 , $s \ 0$, $s \ (s \ 0)$, etc., or as lists of bits—since in a cons-free program these expressions cannot be built unless they already occur in the input—but counting up to limited bounds *can* be achieved by other tricks. By repeatedly simulating a single step of a Turing Machine up to such bounds, Jones shows that any decision problem in EXP^KTIME can be decided using a cons-free program ([12] and Lemma 6).

The core insight in the present paper is that in the presence of non-determinism, an expression of type $\sigma \Rightarrow \tau$ represents a *relation* between expressions of type σ and expressions of type τ rather than a *function*. While the number of functions for a given type is exponential in the order of that type, the number of relations is exponential in the depth of arrows occurring in it. We exploit this (in Lemma 11) by counting up to arbitrarily high numbers using only first-order data. This observation also suggest that by limiting the *arrow depth* rather than the *order of types*, the increase in expressive power disappears (Theorem 3).

Conversely, we also provide an algorithm to compute the output of cons-free programs potentially much faster than the program’s own running time, by using a tableaux to store results. Although similar to Jones’ ideas, our proof style deviates to easily support both non-deterministic and deterministic programs.

1.3 Related work

The creation of programming languages that characterise complexity classes has been a research area since Cobham’s work in the 1960ies, but saw rapid development only after similar advances in the related area of *descriptive complexity* (see, e.g., [10]) in the 1980ies and Bellantoni and Cook’s work on characterisations of P [2] using constraints on recursion in a purely functional language with programs reminiscent of classic recursion theoretic functions. Following Bellantoni and Cook, a number of authors obtained programming languages by constraints on recursion, and under a plethora of names (e.g., *safe*, *tiered* or *ramified* recursion, see [5, 18] for overviews), and this area continues to be active. The main difference with our work is that we consider full recursion in all variables, but place syntactic constraints on the function bodies (both cons-freeness and unitary variables). Also, as in traditional complexity theory we consider decision problems (i.e., what *sets* can be decided by programs), whereas much research in implicit complexity considers functional complexity (i.e., what *functions* can be computed).

¹ The formal definition is slightly more liberal to support easier implementations using pattern-matching, but the ideas remain the same.

Cons-free programs, combined with various limitations on recursion, were introduced by Jones [12], building on ground-breaking work by Goerdt [9,8], and have been studied by a number of authors (see, e.g., [3,4,17,16]). The main difference with our work is that we consider full recursion with full non-determinism, but impose constraints not present in the previous literature.

Characterisation of non-deterministic complexity classes via programming languages remains a largely unexplored area. Bellantoni obtained a characterisation of NP in his dissertation [1] using similar approaches as [2], but at the cost of having a minimisation operator (as in recursion theory), a restriction later removed by Oitavem [19]. A general framework for implicitly characterising a larger hierarchy of non-deterministic classes remains an open problem.

2 A purely functional, non-deterministic, call-by-value programming language

We define a simple call-by-value programming language with explicit non-deterministic choice. This generalises Jones' toy language in [12] by supporting different types and pattern-matching as well as non-determinism. The more permissive language actually *simplifies* proofs and examples, since we do not need to encode all data as boolean lists, and have fewer special cases.

2.1 Syntax

We consider programs defined by the syntax in Figure 2

$$\begin{aligned}
 p &\in \mathbf{Program} ::= \rho_1 \rho_2 \dots \rho_N \\
 \rho &\in \mathbf{Clause} ::= \mathbf{f} \ell_1 \dots \ell_k = s \\
 \ell &\in \mathbf{Pattern} ::= x \mid \mathbf{c} \ell_1 \dots \ell_m \\
 s, t &\in \mathbf{Expr} ::= x \mid \mathbf{c} \mid \mathbf{f} \mid \mathbf{if} s_1 \mathbf{then} s_2 \mathbf{else} s_3 \mid \mathbf{choose} s_1 \dots s_n \mid (s, t) \mid s t \\
 x, y &\in \mathcal{V} ::= \text{identifier} \\
 \mathbf{c} &\in \mathcal{C} ::= \text{identifier disjoint from } \mathcal{V} \quad (\text{we assume } \{\mathbf{true}, \mathbf{false}\} \subseteq \mathcal{C}) \\
 \mathbf{f}, \mathbf{g} &\in \mathcal{D} ::= \text{identifier disjoint from } \mathcal{V} \text{ and } \mathcal{C}
 \end{aligned}$$

Fig. 2. Syntax

We call elements of \mathcal{V} *variables*, elements of \mathcal{C} *data constructors* and elements of \mathcal{D} *defined symbols*. The *root* of a clause $\mathbf{f} \ell_1 \dots \ell_k = s$ is the defined symbol \mathbf{f} . The *main function* \mathbf{f}_1 of the program is the root of ρ_1 . We denote $\text{Var}(s)$ for the set of variables occurring in an expression s . An expression s is *ground* if $\text{Var}(s) = \emptyset$. Application is left-associative, i.e., $s t u$ should be read $(s t) u$.

Definition 1. For expressions s, t , we say that t is a sub-expression of s , notation $s \triangleright t$, if this can be derived using the clauses:

$$\begin{aligned}
 s &\triangleright t \text{ if } s = t \text{ or } s \triangleright t \\
 (s_1, s_2) &\triangleright t \text{ if } s_1 \triangleright t \text{ or } s_2 \triangleright t & \mathbf{if} s_1 \mathbf{then} s_2 \mathbf{else} s_3 \triangleright t \text{ if } s_i \triangleright t \text{ for some } i \\
 s_1 s_2 &\triangleright t \text{ if } s_1 \triangleright t \text{ or } s_2 \triangleright t & \mathbf{choose} s_1 \dots s_n \triangleright t \text{ if } s_i \triangleright t \text{ for some } i
 \end{aligned}$$

Note: the head s of an application $s t$ is not considered a sub-expression of $s t$.

Note that the programs we consider have no pre-defined data structures like integers: these may be encoded using inductive data structures in the usual way.

Example 1. Integers can be encoded as bitstrings of unbounded length: $\mathcal{C} \supseteq \{\text{false}, \text{true}, ::, []\}$. Here, $::$ is considered infix and right-associative, and $[]$ denotes the end of the string. Using little endian, 6 is encoded by $\text{false}::\text{true}::\text{true}::[]$ as well as $\text{false}::\text{true}::\text{true}::\text{false}::\text{false}::[]$. We for instance have $\text{true}::(\text{succ } xs) \sqsupseteq xs$ (for $xs \in \mathcal{V}$). The program below imposes $\mathcal{D} = \{\text{succ}\}$:

$$\begin{aligned} \text{succ } [] &= \text{true}::[] & \text{succ } (\text{false}::xs) &= \text{true}::xs \\ \text{succ } (\text{true}::xs) &= \text{false}::(\text{succ } xs) \end{aligned}$$

2.2 Typing

Programs have explicit simple types without polymorphism, with the usual definition of type order $\text{ord}(\sigma)$; this is formally given in Figure 3.

$\iota \in \mathcal{S} ::= \text{sort identifier}$ $\sigma, \tau \in \text{Type} ::= \iota \mid \sigma \times \tau \mid \sigma \Rightarrow \tau$	$\text{ord}(\iota) = 0 \text{ for } \iota \in \mathcal{S}$ $\text{ord}(\sigma \times \tau) = \max(\text{ord}(\sigma), \text{ord}(\tau))$ $\text{ord}(\sigma \Rightarrow \tau) = \max(\text{ord}(\sigma) + 1, \text{ord}(\tau))$
---	---

Fig. 3. Types and type orders

The (finite) set \mathcal{S} of sorts is used to type atomic data such as bits; we assume $\text{bool} \in \mathcal{S}$. The function arrow \Rightarrow is considered right-associative. Writing κ for a sort or a pair type $\sigma \times \tau$, any type can be uniquely presented in the form $\sigma_1 \Rightarrow \dots \Rightarrow \sigma_m \Rightarrow \kappa$. We will limit interest to *well-typed*, *well-formed* programs:

Definition 2. A program \mathbf{p} is well-typed if there is an assignment \mathcal{F} from $\mathcal{C} \cup \mathcal{D}$ to the set of simple types such that:

- the main function \mathbf{f}_1 is assigned a type $\kappa_1 \Rightarrow \dots \Rightarrow \kappa_M \Rightarrow \kappa$, with $\text{ord}(\kappa_i) = 0$ for $1 \leq i \leq M$ and also $\text{ord}(\kappa) = 0$
- data constructors $\mathbf{c} \in \mathcal{C}$ are assigned a type $\kappa_1 \Rightarrow \dots \Rightarrow \kappa_m \Rightarrow \iota$ with $\iota \in \mathcal{S}$ and $\text{ord}(\kappa_i) = 0$ for $1 \leq i \leq m$
- for all clauses $\mathbf{f} \ell_1 \dots \ell_k = s \in \mathbf{p}$, the following hold:
 - $\text{Var}(s) \subseteq \text{Var}(\mathbf{f} \ell_1 \dots \ell_k)$ and each variable occurs only once in $\mathbf{f} \ell_1 \dots \ell_k$;
 - there exist a type environment Γ mapping $\text{Var}(\mathbf{f} \ell_1 \dots \ell_k)$ to simple types, and a simple type σ , such that both $\mathbf{f} \ell_1 \dots \ell_k : \sigma$ and $s : \sigma$ using the rules in Figure 4; we call σ the type of the clause.

$\frac{}{a : \sigma} \text{ if } a : \sigma \in \Gamma \cup \mathcal{F}$	$\frac{s : \sigma \quad t : \tau}{(s, t) : \sigma \times \tau}$	$\frac{s : \sigma \Rightarrow \tau \quad t : \sigma}{s \ t : \tau}$
$\frac{s_1 : \text{bool} \quad s_2 : \sigma \quad s_3 : \sigma}{\text{if } s_1 \text{ then } s_2 \text{ else } s_3 : \sigma}$	$\frac{s_1 : \sigma \quad \dots \quad s_n : \sigma}{\text{choose } s_1 \dots s_n : \sigma}$	

Fig. 4. Typing (for fixed \mathcal{F} and Γ , see Definition 2)

Note that this definition does not allow for polymorphism: there is a single type assignment \mathcal{F} for the full program. The assignment \mathcal{F} also forces a unique choice for the type environment Γ of variables in each clause. Thus, we may speak of *the* type of an expression in a clause without risk of confusion.

Example 2. The program of Example 1 is typed using $\mathcal{F} = \{\text{false} : \text{bool}, \text{true} : \text{bool}, [] : \text{list}, :: : \text{bool} \Rightarrow \text{list} \Rightarrow \text{list}, \text{succ} : \text{list} \Rightarrow \text{list}\}$. As all argument and output types have order 0, the variable restrictions are satisfied and all clauses can be typed using $\Gamma = \{xs : \text{list}\}$, the program is well-typed.

Definition 3. A program p is well-formed if it is well-typed, and moreover:

- data constructors are always fully applied: for all $c \in \mathcal{C}$ with $c : \kappa_1 \Rightarrow \dots \Rightarrow \kappa_m \Rightarrow \iota \in \mathcal{F}$: if a sub-expression $c \ t_1 \dots t_n$ occurs in any clause, then $n = m$;
- the number of arguments to a given defined symbol is fixed: if $\mathbf{f} \ \ell_1 \dots \ell_k = s$ and $\mathbf{f} \ \ell'_1 \dots \ell'_n = t$ are both in p , then $k = n$; we let $\text{arity}_p(\mathbf{f})$ denote k .

Example 3. The program of Example 1 is well-formed, and $\text{arity}_p(\text{succ}) = 1$.

However, the program would not be well-formed if the clauses below were added, as here the defined symbol `or` does not have a consistent arity.

`id` $x = x$ `or` `true` $x = \text{true}$ `or` `false` $= \text{id}$

Remark 1. Data constructors must (a) have a sort as output type (*not* a pair), and (b) occur only fully applied. This is consistent with typical functional programming languages, where sorts and constructors are declared with a grammar such as:

$sdec \in \text{SortDec} ::= \text{data } \iota = cdec_1 \mid \dots \mid cdec_n$

$cdec \in \text{ConstructorDec} ::= c \ \sigma_1 \ \dots \ \sigma_m$

In addition, we require that the arguments to data constructors have type order 0. This is not standard in functional programming, but is the case in [12]. We limit interest to such constructors because, practically, these are the only ones which can be used in a *cons-free* program (as we will discuss in Section 3).

Definition 4. A program has data order K if all clauses can be typed using type environments Γ such that, for all $x : \sigma \in \Gamma$: $\text{ord}(\sigma) \leq K$.

Example 4. We consider a higher-order program, operating on the same data constructors as Example 1; however, now we encode numbers using *functions*:

```
fsucc F [] = if F [] then set F [] false else set F [] true
fsucc F xs = if F xs then fsucc (set F xs false) (tl xs)
               else set F xs true
set F val xs ys = if eqlen xs ys then val else F ys
tl (x::xs) = xs      eqlen (x::xs) (y::ys) = eqlen xs ys
eqlen [] [] = true   eqlen xs ys = false
```

Only one typing is possible, with $\text{fsucc} : (\text{list} \Rightarrow \text{bool}) \Rightarrow \text{list} \Rightarrow \text{list} \Rightarrow \text{bool}$; therefore, F is always typed $\text{list} \Rightarrow \text{bool}$ —which has type order 1—and all other variables with a type of order 0. Thus, this program has data order 1.

To explain the program: we use boolean lists as *unary* numbers of a limited size; assuming that (a) F represents a bitstring of length $N + 1$, and (b) lst has length N , the successor of F (modulo wrapping) is obtained by $\text{fsucc } F \ lst$.

2.3 Semantics

Like Jones, our language has a closure-based call-by-value semantics. We let data expressions, values and environments be defined by the grammar in Figure 5.

$d, b \in \mathbf{Data} ::= c\ d_1 \cdots d_m \mid (d, b)$	Instantiation:
$v, w \in \mathbf{Value} ::= d \mid (v, w) \mid f\ v_1 \cdots v_n$ $(n < \mathbf{arity}_p(f))$	$x\gamma := \gamma(x)$
$\gamma, \delta \in \mathbf{Env} ::= \mathcal{V} \rightarrow \mathbf{Value}$	$(c\ \ell_1 \cdots \ell_n)\gamma := c\ (\ell_1\gamma) \cdots (\ell_n\gamma)$

Fig. 5. Data expressions, values and environments

Let $\mathbf{dom}(\gamma)$ denote the domain of an environment (partial function) γ . Note that values are ground expressions, and we only use well-typed values with fully applied data constructors. To every pattern ℓ and environment γ with $\mathbf{dom}(\gamma) \supseteq \mathbf{Var}(\ell)$, we associate a value $\ell\gamma$ by instantiation in the obvious way, see Figure 5.

Note that, for every value v and pattern ℓ , there is at most one environment γ with $\ell\gamma = v$. We say that an expression $\mathbf{f}\ s_1 \cdots s_n$ *instantiates* the left-hand side of a clause $\mathbf{f}\ \ell_1 \cdots \ell_k$ if $n = k$ and there is an environment γ with each $s_i = \ell_i\gamma$.

Both input and output to the program are data expressions. If \mathbf{f}_1 has type $\kappa_1 \Rightarrow \dots \Rightarrow \kappa_M \Rightarrow \kappa$, we can think of the program as calculating a function $\llbracket \mathbf{p} \rrbracket(d_1, \dots, d_M)$ from M input data arguments to an output data expression.

Expression and program evaluation are given by the rules in Figure 6. Since, in [Call], there is at most one suitable γ , the only source of non-determinism is the **choose** operator. Programs without this operator are called *deterministic*. By contrast, we may refer to a *non-deterministic* program as one which is not explicitly required to be deterministic, so which may or may not contain **choose**.

Example 5. For the program from Example 1, $\llbracket \mathbf{p} \rrbracket(\mathbf{true}::\mathbf{false}::\mathbf{true}::[]) \mapsto \mathbf{false}::\mathbf{true}::\mathbf{true}::[]$, giving $5 + 1 = 6$. In the program $\mathbf{f}_1\ x\ y = \mathbf{choose}\ x\ y$, we can both derive $\llbracket \mathbf{p} \rrbracket(\mathbf{true}, \mathbf{false}) \mapsto \mathbf{true}$ and $\llbracket \mathbf{p} \rrbracket(\mathbf{true}, \mathbf{false}) \mapsto \mathbf{false}$.

The language is easily seen to be Turing-complete unless further restrictions are imposed. In order to assuage any fears on whether the complexity-theoretic characterisations we obtain are due to brittle design choices, we add some remarks.

Remark 2. We have omitted some constructs common to even some toy pure functional languages, but these are in general simple syntactic sugar that can be readily expressed by the existing constructs in the language, even in the presence of non-determinism. For instance, a let-binding $\mathbf{let}\ x = s_1\ \mathbf{in}\ s_2$ can be straightforwardly encoded by a function call in a pure call-by-value setting (replacing $\mathbf{let}\ x = s_1\ \mathbf{in}\ s_2$ by $\mathbf{helper}\ s_1$ and adding a clause $\mathbf{helper}\ x = s_2$).

Remark 3. We do not require the clauses of a function definition to exhaust all possible patterns. For instance, it is possible to have a clause $\mathbf{f}\ \mathbf{true} = \dots$ without a clause for $\mathbf{f}\ \mathbf{false}$. Thus, a program has zero or more values.

Expression evaluation:	
[Instance]: $\frac{}{\mathbf{p}, \gamma \vdash x \rightarrow \gamma(x)}$	[Function]: $\frac{\mathbf{p} \vdash^{\text{call}} \mathbf{f} \rightarrow w}{\mathbf{p}, \gamma \vdash \mathbf{f} \rightarrow w}$ for $\mathbf{f} \in \mathcal{D}$
[Constructor]: $\frac{\mathbf{p}, \gamma \vdash s_1 \rightarrow b_1 \quad \cdots \quad \mathbf{p}, \gamma \vdash s_m \rightarrow b_m}{\mathbf{p}, \gamma \vdash \mathbf{c} \ s_1 \cdots s_m \rightarrow \mathbf{c} \ b_1 \cdots b_m}$	
[Pair]: $\frac{\mathbf{p}, \gamma \vdash s \rightarrow v \quad \mathbf{p}, \gamma \vdash t \rightarrow w}{\mathbf{p}, \gamma \vdash (s, t) \rightarrow (v, w)}$	
[Choice]: $\frac{\mathbf{p}, \gamma \vdash s_i \rightarrow w}{\mathbf{p}, \gamma \vdash \text{choose } s_1 \cdots s_n \rightarrow w}$ for $1 \leq i \leq n$	
[Conditional]: $\frac{\mathbf{p}, \gamma \vdash s_1 \rightarrow d \quad \mathbf{p}, \gamma \vdash^{\text{if}} d, s_2, s_3 \rightarrow w}{\mathbf{p}, \gamma \vdash \text{if } s_1 \text{ then } s_2 \text{ else } s_3 \rightarrow w}$	
[If-True]: $\frac{\mathbf{p}, \gamma \vdash s_2 \rightarrow w}{\mathbf{p}, \gamma \vdash^{\text{if}} \text{true}, s_2, s_3 \rightarrow w}$	[If-False]: $\frac{\mathbf{p}, \gamma \vdash s_3 \rightarrow w}{\mathbf{p}, \gamma \vdash^{\text{if}} \text{false}, s_2, s_3 \rightarrow w}$
[Appl]: $\frac{\mathbf{p}, \gamma \vdash s \rightarrow \mathbf{f} \ v_1 \cdots v_n \quad \mathbf{p}, \gamma \vdash t \rightarrow v_{n+1} \quad \mathbf{p} \vdash^{\text{call}} \mathbf{f} \ v_1 \cdots v_{n+1} \rightarrow w}{\mathbf{p}, \gamma \vdash s \ t \rightarrow w}$	
[Closure]: $\frac{}{\mathbf{p} \vdash^{\text{call}} \mathbf{f} \ v_1 \cdots v_n \rightarrow \mathbf{f} \ v_1 \cdots v_n}$ if $n < \text{arity}_{\mathbf{p}}(\mathbf{f})$	
[Call]: $\frac{\mathbf{p}, \gamma \vdash s \rightarrow w}{\mathbf{p} \vdash^{\text{call}} \mathbf{f} \ v_1 \cdots v_k \rightarrow w}$ if $\mathbf{f} \ \ell_1 \cdots \ell_k = s$ is the first clause in \mathbf{p} such that $\mathbf{f} \ v_1 \cdots v_k$ instantiates $\mathbf{f} \ \ell_1 \cdots \ell_k$, and $\text{dom}(\gamma) = \text{Var}(\mathbf{f} \ \ell_1 \cdots \ell_k)$ and each $v_i = \ell_i \gamma$	
Program execution:	
$\frac{\mathbf{p}, [x_1 := d_1, \dots, x_M := d_M] \vdash \mathbf{f}_1 \ x_1 \cdots x_M \rightarrow b}{\llbracket \mathbf{p} \rrbracket(d_1, \dots, d_M) \mapsto b}$	

Fig. 6. Call-by-value semantics

Data order versus program order. We have followed Jones in considering *data order* as the variable for increasing complexity. However, an alternative choice—which turns out to streamline our proofs—is *program order*, which considers the type order of the function symbols. Fortunately, these notions are closely related; barring unused symbols, $\langle \text{program order} \rangle = \langle \text{data order} \rangle + 1$.

More specifically, we have the following result:

Lemma 1. *For every well-formed program \mathbf{p} with data order K , there is a well-formed program \mathbf{p}' such that $\llbracket \mathbf{p} \rrbracket(d_1, \dots, d_M) \mapsto b$ iff $\llbracket \mathbf{p}' \rrbracket(d_1, \dots, d_M) \mapsto b$ for any b_1, \dots, b_M, d and: (a) all defined symbols in \mathbf{p}' have a type $\sigma_1 \Rightarrow \dots \Rightarrow \sigma_m \Rightarrow \kappa$ such that both $\text{ord}(\sigma_i) \leq K$ for all i and $\text{ord}(\kappa) \leq K$, and (b) in all clauses, all sub-expressions of the right-hand side have a type of order $\leq K$ as well.*

Proof (Sketch). \mathbf{p}' is obtained from \mathbf{p} through the following successive changes:

1. Replace any clause $\mathbf{f} \ell_1 \cdots \ell_k = s$ where $s : \sigma \Rightarrow \tau$ with $\text{ord}(\sigma \Rightarrow \tau) = K + 1$, by $\mathbf{f} \ell_1 \cdots \ell_k x = s x$ for a fresh x . Repeat until no such clauses remain.
2. In any clause $\mathbf{f} \ell_1 \cdots \ell_k = s$, replace all sub-expressions $(\text{choose } s_1 \cdots s_m) t_1 \cdots t_n$ or $(\text{if } s_1 \text{ then } s_2 \text{ else } s_3) t_1 \cdots t_n$ of s with $n > 0$ by $\text{choose } (s_1 t_1 \cdots t_n) \cdots (s_m t_1 \cdots t_n)$ or $\text{if } s_1 \text{ then } (s_2 t_1 \cdots t_n) \text{ else } (s_3 t_1 \cdots t_n)$ respectively.
3. In any clause $\mathbf{f} \ell_1 \cdots \ell_k = s$, if s has a sub-expression $t = \mathbf{g} s_1 \cdots s_n$ with $\mathbf{g} : \sigma_1 \Rightarrow \dots \Rightarrow \sigma_n \Rightarrow \tau$ such that $\text{ord}(\tau) \leq K$ but $\text{ord}(\sigma_i) > K$ for some i , then replace t by a fresh symbol \perp_τ . Repeat until no such sub-expressions remain, then add clauses $\perp_\tau = \perp_\tau$ for the new symbols.
4. If there exists $\mathbf{f} : \sigma_1 \Rightarrow \dots \Rightarrow \sigma_m \Rightarrow \kappa \in \mathcal{F}$ with $\text{ord}(\kappa) > K$ or $\text{ord}(\sigma_i) > K$ for some i , then remove the symbol \mathbf{f} and all clauses with root \mathbf{f} .

The key observation is that if the derivation for $\llbracket \mathbf{p} \rrbracket(d_1, \dots, d_M) \mapsto b$ uses some $\mathbf{f} s_1 \cdots s_n : \sigma$ with $\text{ord}(\sigma) \leq K$ but $s_i : \tau$ with $\text{ord}(\tau) > K$, then there is a variable with type order $> K$. Thus, if a clause introduces such an expression, either the clause is never used, or the expression occurs beneath an **if** or **choose** and is never selected; it may be replaced with a symbol whose only rule is unusable. This also justifies step 1; for step 4, only unusable clauses are removed.

(See Appendix A for the complete proof.) \square

Example 6. The following program has data order 0, but clauses of functional type; **fst** and **snd** have output type $\text{nat} \Rightarrow \text{nat}$ of order 1. The program is changed by replacing the last two clauses by **fst** $x y = \text{const } x y$ and **snd** $x y = \text{id } y$.

```

start xs ys = choose (fst xs ys) (snd xs ys)
const x y = x      fst x = const x
id x = x           snd x = id

```

3 Cons-free programs

Jones defines a cons-free program as one where the list constructor $::$ does not occur in any clause. In our setting (where more constructors are in principle admitted), this translates to disallowing non-constant data constructors from being introduced in the right-hand side of a clause. We define:

Definition 5. A program \mathbf{p} is *cons-free* if all clauses in \mathbf{p} are cons-free. A clause $\mathbf{f} \ell_1 \cdots \ell_k = s$ is *cons-free* if for all $s \geq t$: if $t = \mathbf{c} s_1 \cdots s_m$ with $\mathbf{c} \in \mathcal{C}$, then t is a data expression or $\ell_i \geq t$ for some i .

Example 7. Example 1 is not cons-free, due to the second and third clause (the first clause is cons-free). Examples 4 and 6 are both cons-free.

The key property of cons-free programming is that no *new* data structures can be created during program execution. Formally, in a derivation tree with root $\llbracket \mathbf{p} \rrbracket(d_1, \dots, d_M) \mapsto b$, all data values (including b) are in the set $\mathcal{B}_{d_1, \dots, d_M}^{\mathbf{p}}$:

Definition 6. Let $\mathcal{B}_{d_1, \dots, d_M}^p := \{d \in \text{Data} \mid \exists i[d_i \triangleright d] \vee \exists(\mathbf{f} \ell = s) \in p[s \triangleright d]\}$.

$\mathcal{B}_{d_1, \dots, d_M}^p$ is a set of data expressions closed under \triangleright , with a linear number of elements in the size of d_1, \dots, d_M (for fixed p). The property that no new data is created during execution is formally expressed by the following lemma.

Lemma 2. Let p be a cons-free program, and suppose that $\llbracket p \rrbracket(d_1, \dots, d_M) \mapsto b$ is obtained by a derivation tree T . Then for all statements $p, \gamma \vdash s \rightarrow w$ or $p, \gamma \vdash^{\text{if}} b', s_1, s_2 \rightarrow w$ or $p \vdash^{\text{call}} \mathbf{f} v_1 \dots v_n \rightarrow w$ in T , and all expressions t such that (a) $w \triangleright t$, (b) $b' \triangleright t$, (c) $\gamma(x) \triangleright t$ for some x or (d) $v_i \triangleright t$ for some i : if t has the form $\mathbf{c} b_1 \dots b_m$ with $\mathbf{c} \in \mathcal{C}$, then $t \in \mathcal{B}_{d_1, \dots, d_M}^p$.

That is, any data expression in the derivation tree of $\llbracket p \rrbracket(d_1, \dots, d_M) \mapsto b$ (including occurrences as a sub-expression of other values) is also in $\mathcal{B}_{d_1, \dots, d_M}^p$.

Proof (Sketch). Induction on the form of T , assuming that for a statement under consideration, (1) the requirements on γ and the v_i are satisfied, and (2) γ maps expressions $t \trianglelefteq s, s_1, s_2$ to elements of $\mathcal{B}_{d_1, \dots, d_M}^p$ if $t = \mathbf{c} t_1 \dots t_m$ with $\mathbf{c} \in \mathcal{C}$.

(See Appendix B for the complete proof.) \square

Note that Lemma 2 implies that the program result b is in $\mathcal{B}_{d_1, \dots, d_M}^p$. Recall also Remark 1: if we had admitted constructors with higher-order argument types, then Lemma 2 shows that they are never used, since any constructor appearing in a derivation for $\llbracket p \rrbracket(d_1, \dots, d_M) \mapsto b$ must already occur in the (data!) input.

4 Turing Machines, decision problems and complexity

We assume familiarity with the standard notions of Turing Machines and complexity classes (see, e.g., [20, 11, 21]); in this section, we fix the notation we use.

4.1 (Deterministic) Turing Machines

Turing Machines (TMs) are triples (A, S, T) where A is a finite set of tape symbols such that $A \supseteq \{0, 1, \perp\}$, $S \supseteq \{\text{start}, \text{accept}, \text{reject}\}$ is a finite set of states, and T is a finite set of transitions (i, r, w, d, j) with $i \in S \setminus \{\text{accept}, \text{reject}\}$ (the *original state*), $r \in A$ (the *read symbol*), $w \in A$ (the *written symbol*), $d \in \{\text{L}, \text{R}\}$ (the *direction*), and $j \in S$ (the *result state*). We sometimes denote this transition as $i \xrightarrow{r/w \ d} j$.

A *deterministic* Turing Machine is a TM such that every pair (i, r) with $i \in S \setminus \{\text{accept}, \text{reject}\}$ and $r \in A$ is associated with exactly one transition (i, r, w, d, j) . Every TM in this paper has a single, right-infinite tape.

A *valid tape* is an element t of $A^{\mathbb{N}}$ with $t(p) \neq \perp$ for only finitely many p . A *configuration* is a triple (t, p, s) with t a valid tape, $p \in \mathbb{N}$ and $s \in S$. The transitions T induce a relation \Rightarrow between configurations in the obvious way.

4.2 Decision problems

A *decision problem* is a set $X \subseteq \{0, 1\}^+$. A deterministic TM *decides* X if for any $x \in \{0, 1\}^+$: $x \in X$ iff $\sqcup x_1 \dots x_n \sqcup \dots, 0, \mathbf{start} \Rightarrow^* (t, i, \mathbf{accept})$ for some t, i , and $(\sqcup x_1 \dots x_n \sqcup \dots, 0, \mathbf{start}) \Rightarrow^* (t, i, \mathbf{reject})$ iff $x \notin X$. Thus, the TM halts on all inputs, ending in **accept** or **reject** depending on whether $x \in X$.

If $h : \mathbb{N} \rightarrow \mathbb{N}$ is a function, a deterministic TM *runs in time* $\lambda n. h(n)$ if for all $n \in \mathbb{N} \setminus \{0\}$ and $x \in \{0, 1\}^n$: any evaluation starting in $(\sqcup x_1 \dots x_n \sqcup \dots, 0, \mathbf{start})$ ends in the **accept** or **reject** state in at most $h(n)$ transitions.

4.3 Complexity and the EXPTIME hierarchy

We define classes of decision problem based on the *time* needed to accept them.

Definition 7. Let $h : \mathbb{N} \rightarrow \mathbb{N}$ be a function. Then, $\text{TIME}(h(n))$ is the set of all $X \subseteq \{0, 1\}^+$ such that there exist $a > 0$ and a deterministic TM running in time $\lambda n. a \cdot h(n)$ that decides X .

By design, $\text{TIME}(h(n))$ is closed under \mathcal{O} : $\text{TIME}(h(n)) = \text{TIME}(\mathcal{O}(h(n)))$.

Definition 8. For $K, n \geq 0$, let $\exp_2^0(n) = n$ and $\exp_2^{K+1}(n) = \exp_2^K(2^n) = 2^{\exp_2^K(n)}$. For $K \geq 0$, define $\text{EXP}^K \text{TIME} \triangleq \bigcup_{a, b \in \mathbb{N}} \text{TIME}(\exp_2^K(an^b))$.

Since for every polynomial h , there are $a, b \in \mathbb{N}$ such that $h(n) \leq a \cdot n^b$ for all $n > 0$, we have $\text{EXP}^0 \text{TIME} = \text{P}$ and $\text{EXP}^1 \text{TIME} = \text{EXP}$ (where **EXP** is the usual complexity class of this name, see e.g., [20, Ch. 20]). In the literature, **EXP** is sometimes called **EXPTIME** or **DEXPTIME** (e.g., in the celebrated proof that ML typability is complete for **DEXPTIME** [13]). Using the Time Hierarchy Theorem [21], it is easy to see that $\text{P} = \text{EXP}^0 \text{TIME} \subsetneq \text{EXP}^1 \text{TIME} \subsetneq \text{EXP}^2 \text{TIME} \subsetneq \dots$.

Definition 9. The set **ELEMENTARY** of elementary-time computable languages is $\bigcup_{K \in \mathbb{N}} \text{EXP}^K \text{TIME}$.

4.4 Decision problems and programs

To solve decision problems by (cons-free) programs, we will consider programs with constructors **true**, **false** of type **bool**, \square of type **list** and $::$ of type $\text{bool} \Rightarrow \text{list} \Rightarrow \text{list}$, and whose main function f_1 has type $\text{list} \Rightarrow \text{bool}$.

Definition 10. We define:

- A program p accepts $a_1 a_2 \dots a_n \in \{0, 1\}^*$ if $\llbracket p \rrbracket(\overline{a_1} :: \dots :: \overline{a_n}) \mapsto \mathbf{true}$, where $\overline{a_i} = \mathbf{true}$ if $a_i = 1$ and $\overline{a_i} = \mathbf{false}$ otherwise.
- The set accepted by program p is $\{a \in \{0, 1\}^* \mid p \text{ accepts } a\}$.

Although we focus on programs of this form, our proofs will allow for arbitrary input and output—with the limitation (as guaranteed by the rule for program execution) that both are data. This makes it possible to for instance consider decision problems on a larger input alphabet without needing encodings.

Example 8. The two-line program with clauses `even [] = true` and `even (x::xs) = if x then false else true` accepts the problem $\{x \in \{0,1\}^* \mid x \text{ is a bitstring representing an even number (following Example 1)}\}$.

We will sometimes speak of the input size, defined by:

Definition 11. The size of a list of data expressions d_1, \dots, d_M is $\sum_{i=1}^M \text{size}(d_i)$, where $\text{size}(\text{c } b_1 \dots b_m)$ is defined as $1 + \sum_{i=1}^m \text{size}(b_i)$.

5 Deterministic characterisations

As a basis, we transfer Jones' basic result on *time* classes to our more general language. That is, we obtain the first line of the first table in Figure 1.

	data order 0	data order 1	data order 2	data order 3	...
cons-free	P =	EXP =	EXP ² TIME	EXP ³ TIME	...
deterministic	EXP ⁰ TIME	EXP ¹ TIME			

To show that deterministic cons-free programs of data order K characterise $\text{EXP}^K \text{TIME}$ it is necessary to prove two things:

1. if $h(n) \leq \exp_2^K(a \cdot n^b)$ for all n , then for every deterministic Turing Machine M running in $\text{TIME}(h(n))$, there is a deterministic, cons-free program with data order at most K , which accepts $x \in \{0,1\}^+$ if and only if M does;
2. for every deterministic cons-free program \mathbf{p} with data order K , there is a deterministic algorithm operating in $\text{TIME}(\exp_2^K(a \cdot n^b))$ for some a, b which, given input expressions d_1, \dots, d_M , determines b such that $\llbracket \mathbf{p} \rrbracket(d_1, \dots, d_M) \mapsto b$ (if such b exists). Like Jones [12], we assume our algorithms are implemented on a sufficiently expressive Turing-equivalent machine like the RAM.

We will show part (1) in Section 5.1, and part (2) in Section 5.2.

5.1 Simulating TMs using deterministic cons-free programs

Let $M := (A, S, T)$ be a deterministic Turing Machine running in time $\lambda n.h(n)$. Like Jones, we start by assuming that we have a way to represent the numbers $0, \dots, h(n)$ as expressions, along with successor and predecessor operators and checks for equality. Our simulation uses the following data constructors

- `true : bool`, `false : bool`, `[] : list` and `:: : bool \Rightarrow list \Rightarrow list` as discussed in Section 4.4;
- `a : symbol` for $a \in A$ (writing `B` for the blank symbol), `L, R : direc` and `s : state` for $s \in S$;
- `action : symbol \Rightarrow direc \Rightarrow state \Rightarrow trans`; and
- `end : state \Rightarrow trans`.

The rules to simulate the machine are given in Figure 7.

```

run cs = test (state cs [h(|cs|)])
test accept = true          transition i r = action w d j   for all  $i \xrightarrow{r/w d} j \in T$ 
test reject = false         transition i x = end i           for  $i \in \{\text{accept}, \text{reject}\}$ 

state cs [n] = if [n = 0] then start else get3 (transat cs [n - 1])
transat cs [n] = transition (state cs [n]) (tapesymb cs [n])

get1 (action x y z) = x      get1 (end x) = B
get2 (action x y z) = y      get2 (end x) = R
get3 (action x y z) = z      get3 (end x) = x

tapesymb cs [n] = tape cs [n] (pos cs [n])

tape cs [n] [p] = if [n = 0] then inputtape cs [p]
                  else tapehelp cs [n] [p] (pos cs [n - 1])
tapehelp cs [n] [p] [i] = if [p = i] then get1 (transat cs [n - 1])
                          else tape cs [n - 1] [p]

pos cs [n] = if [n = 0] then [0] else adjust cs (pos cs [n - 1]) (get2 (transat cs [n - 1]))
adjust cs [p] L = [p - 1]    adjust cs [p] R = [p + 1]

inputtape cs [p] = if [p = 0] then B else nth cs [p - 1]
nth [] [p] = B
nth (x::xs) [p] = if [p = 0] then bit x else nth xs [p - 1]    bit true = 1
                                                                bit false = 0

```

Fig. 7. Simulating a deterministic Turing Machine (A, S, T)

Types of defined symbols are easily derived. The intended meaning is that **state** $cs [n]$, for cs the input list and $[n]$ a number in $\{0, \dots, h(|cs|)\}$, returns the state of the machine at time $[n]$; **pos** $cs [n]$ returns the position of the reader at time $[n]$, and **tape** $cs[n] [p]$ the symbol at time $[n]$ and position $[p]$.

Clearly, the program is highly exponential, even when $h(|cs|)$ is polynomial, since the same expressions are repeatedly evaluated. This apparent contradiction is not problematic: we do not claim that all cons-free programs with data order 0 (say) have a derivation tree of at most polynomial size. Rather, as we will see in Section 5.2, we can find their *result* in polynomial time by essentially using a caching mechanism to avoid reevaluating the same expression.

What remains is to simulate numbers and counting. For a machine running in $\text{TIME}(h(n))$, it suffices to find a value $[i]$ representing i for all $i \in \{0, \dots, h(n)\}$ and cons-free clauses to calculate predecessor and successor functions and to perform zero and equality checks. This is given by a $(\lambda n. h(n) + 1)$ -counting module. This defines, for a given input list cs of length n , a set of values \mathcal{A}_π^n to represent numbers and functions **seed** $_\pi$, **pred** $_\pi$ and **zero** $_\pi$ such that (a) **seed** $_\pi$ cs evaluates to a value which represents $h(n)$, (b) if v represents a number k , then **pred** $_\pi$ cs v evaluates to a value which represents $k - 1$, and (c) **zero** $_\pi$ cs v evaluates to **true** or **false** depending on whether v represents 0. Formally:

Definition 12 (Adapted from [12]). For $P : \mathbb{N} \rightarrow \mathbb{N} \setminus \{0\}$, a P -counting module is a tuple $C_\pi = (\alpha_\pi, \mathcal{D}_\pi, \mathcal{A}_\pi, \langle \cdot \rangle_\pi, \mathbf{p}_\pi)$ such that:

- α_π is a type (this will be the type of numbers);

- \mathcal{D}_π is a set of defined symbols disjoint from $\mathcal{C}, \mathcal{D}, \mathcal{V}$, containing symbols seed_π , pred_π and zero_π , with types $\text{seed}_\pi : \text{list} \Rightarrow \alpha_\pi$, $\text{pred}_\pi : \text{list} \Rightarrow \alpha_\pi \Rightarrow \alpha_\pi$ and $\text{zero}_\pi : \text{list} \Rightarrow \alpha_\pi \Rightarrow \text{bool}$;
- for $n \in \mathbb{N}$, \mathcal{A}_π^n is a set of values of type α_π , all built over $\mathcal{C} \cup \mathcal{D}_\pi$ (this is the set of values used to represent numbers);
- for $n \in \mathbb{N}$, $\langle \cdot \rangle_\pi^n$ is a total function from \mathcal{A}_π^n to \mathbb{N} ;
- \mathbf{p}_π is a list of cons-free clauses on the symbols in \mathcal{D}_π , such that, for all lists $cs : \text{list} \in \text{Data}$ with length n :
 - there is a unique value v such that $\mathbf{p}_\pi \vdash^{\text{call}} \text{seed}_\pi cs \rightarrow v$;
 - if $\mathbf{p}_\pi \vdash^{\text{call}} \text{seed}_\pi cs \rightarrow v$, then $v \in \mathcal{A}_\pi^n$ and $\langle v \rangle_\pi^n = P(n) - 1$;
 - if $v \in \mathcal{A}_\pi^n$ and $\langle v \rangle_\pi^n = i > 0$, then there is a unique value w such that $\mathbf{p}_\pi \vdash^{\text{call}} \text{pred}_\pi cs v \rightarrow w$; we have $w \in \mathcal{A}_\pi^n$ and $\langle w \rangle_\pi^n = i - 1$;
 - for $v \in \mathcal{A}_\pi^n$ with $\langle v \rangle_\pi^n = i$: $\mathbf{p}_\pi \vdash^{\text{call}} \text{zero}_\pi cs v \rightarrow \text{true}$ if and only if $i = 0$, and $\mathbf{p}_\pi \vdash^{\text{call}} \text{zero}_\pi cs v \rightarrow \text{false}$ if and only if $i > 0$.

It is easy to see how a P -counting module can be plugged into the program of Figure 7. We only lack successor and equality functions, which are easily defined:

```

succ $_\pi$  cs i = sc $_\pi$  cs (seed $_\pi$  cs) i
sc $_\pi$  cs j i = if equal $_\pi$  cs (pred $_\pi$  cs j) i then j else sc cs (pred $_\pi$  cs j) i
equal $_\pi$  cs i j = if zero $_\pi$  cs i then zero $_\pi$  cs j
                  else if zero $_\pi$  cs j then false
                  else equal $_\pi$  cs (pred $_\pi$  cs i) (pred $_\pi$  cs j)

```

Since the clauses in Figure 7 are cons-free and have data order 0, we obtain:

Lemma 3. *Let x be a decision problem which can be decided by a deterministic TM running in $\text{TIME}(h(n))$. If there is a cons-free $(\lambda n. h(n) + 1)$ -counting module C_π with data order K , then x is accepted by a cons-free program with data order K ; the program is deterministic if the counting module is.*

Proof. By the argument given above. □

The obvious difficulty is the restriction to cons-free clauses: we cannot simply construct a new number type, but will have to represent numbers using only sub-expressions of the input list cs , and constant data expressions.

Example 9. We consider a P -counting module C_x where $P(n) = 3 \cdot (n + 1)^2$. Let $\alpha_x := \text{list} \times \text{list} \times \text{list}$ and for given n , let $\mathcal{A}_x^n := \{(d_0, d_1, d_2) \mid d_0 \text{ is a list of length } \leq 2 \text{ and } d_1, d_2 \text{ are lists of length } \leq n\}$. Writing $|x_1 :: \dots :: x_k :: []| = k$, let $\langle (d_0, d_1, d_2) \rangle_x^n := |d_0| \cdot (n + 1)^2 + |d_1| \cdot (n + 1) + |d_2|$. Essentially, we consider 3-digit numbers $i_0 i_1 i_2$ in base $n + 1$, with each i_j represented by a list. \mathbf{p}_x is:

```

seed $_x$  cs = (false::false::[], cs, cs)
pred $_x$  cs (x $_0$ , x $_1$ , y::ys) = (x $_0$ , x $_1$ , ys)   zero $_x$  cs (x $_0$ , x $_1$ , y::ys) = false
pred $_x$  cs (x $_0$ , y::ys, []) = (x $_0$ , ys, cs)     zero $_x$  cs (x $_0$ , y::ys, []) = false
pred $_x$  cs (y::ys, [], []) = (ys, cs, cs)      zero $_x$  cs (y::ys, [], []) = false
pred $_x$  cs ([], [], []) = ([], [], [])         zero $_x$  cs ([], [], []) = true

```


If $cs = \text{true}::\text{false}::\text{true}::[]$, one value in \mathcal{A}_x^3 is $v = (\text{false}::[], \text{false}::\text{true}::[], [])$, which is mapped to the number $1 \cdot 4^2 + 2 \cdot 4 + 0 = 24$. Then $\mathbf{p}_x \vdash^{\text{call}} \text{pred}_x cs v \rightarrow w := (\text{false}::[], \text{true}::[], cs)$, which is mapped to $1 \cdot 4^2 + 1 \cdot 4 + 3 = 23$ as desired.

Example 9 suggests a systematic way to create polynomial counting modules.

Lemma 4. *For any $a, b \in \mathbb{N} \setminus \{0\}$, there is a $(\lambda n.a \cdot (n+1)^b)$ -counting module $C_{\langle a,b \rangle}$ with data order 0.*

Proof (Sketch). A straightforward generalisation of Example 9

(See Appendix A for the complete proof.) \square

By increasing type orders, we can obtain an exponential increase of magnitude.

Lemma 5. *If there is a P -counting module C_π of data order K , then there is a $(\lambda n.2^{P(n)})$ -counting module $C_{e[\pi]}$ of data order $K+1$.*

Proof (Sketch). Let $\alpha_{e[\pi]} := \alpha_\pi \Rightarrow \text{bool}$; then $\text{ord}(\alpha_{e[\pi]}) \leq K+1$. A number i with bit representation $b_0 \dots b_{P(n)-1}$ (with b_0 the most significant digit) is represented by a value v such that, for w with $\langle w \rangle_\pi = i$: $\mathbf{p}_{e[\pi]} \vdash^{\text{call}} v w \rightarrow \text{true}$ iff $b_i = 1$, and $\mathbf{p}_{e[\pi]} \vdash^{\text{call}} v w \rightarrow \text{false}$ iff $b_i = 0$. We use the clauses of Figure 8.

```

seede[π] cs x = true
zeroe[π] cs F = zhlpe[π] cs F (seedπ cs)
zhlpe[π] cs F k = if F k then false
                  else if zeroπ cs k then true
                  else zhlpe[π] cs F (predπ cs k)
prede[π] cs F = phlpe[π] cs F (seedπ cs)
phlpe[π] cs F k = if F k then flipe[π] cs F k
                  else if zeroπ cs k then seede[π] cs
                  else phlpe[π] cs (flipe[π] cs F k) (predπ cs k)
flipe[π] cs F k i = if equalπ cs k i then not (F i) else F i
not b = if b then false else true

```

Fig. 8. The clauses used in $\mathbf{p}_{e[\pi]}$, extending \mathbf{p}_π with an exponential step.

We also include all clauses in \mathbf{p}_π . Here, note that a bitstring $b_0 \dots b_m$ represents 0 if each $b_i = 0$, and that the predecessor of $b_0 \dots b_i 10 \dots 0$ is $b_0 \dots b_i 01 \dots 1$.

(See Appendix A for the complete proof.) \square

Combining these results, we obtain:

Lemma 6. *Every decision problem in $\text{EXP}^K \text{TIME}$ is accepted by a deterministic cons-free program with data order K .*

Proof. A decision problem is in $\text{EXP}^K \text{TIME}$ if it is decided by a deterministic TM operating in time $\exp_2^K(a \cdot n^b)$ for some a, b . By Lemma 3, it therefore suffices if there is a Q -counting module for some $Q \geq \lambda n. \exp_2^K(a \cdot n^b) + 1$, with data order K . Certainly $Q(n) := \exp_2^K(a \cdot (n+1)^b)$ is large enough. By Lemma 4, there is a $(\lambda n.a \cdot (n+1)^b)$ -counting module $C_{\langle a,b \rangle}$ with data order 0. Applying Lemma 5 K times, we obtain the required Q -counting module $C_{e[\dots[e[\langle a,b \rangle]]]}$. \square

Remark 4. Our definition of a counting module significantly differs from the one in [12], for example by representing numbers as *values* rather than *expressions*, introducing the sets \mathcal{A}_π^n and imposing evaluation restrictions. The changes enable an easy formulation of the non-deterministic counting module in Section 6.

5.2 Simulating deterministic cons-free programs using an algorithm

We now turn to the second part of characterisation: that every decision problem solved by a deterministic cons-free program of data order K is in $\text{EXP}^K \text{TIME}$. We give an algorithm which determines the result of a fixed program (if any) on a given input in $\text{TIME}(\exp_2^K(a \cdot n^b))$ for some a, b . The algorithm is designed to extend easily to the non-deterministic characterisations in subsequent settings.

Key idea. The principle of our algorithm is easy to explain when variables have data order 0. Using Lemma 2, all such variables must be instantiated by (tuples of) elements of $\mathcal{B}_{d_1, \dots, d_M}^p$, of which there are only polynomially many in the input size. Thus, we can make a comprehensive list of all expressions that might occur as the left-hand side of a [Call] in the derivation tree. Now we can go over the list repeatedly, filling in reductions to trace a top-down derivation of the tree.

In the higher-order setting, there are infinitely many possible values; for example, if $\text{id} : \text{bool} \Rightarrow \text{bool}$ has arity 1 and $\text{g} : (\text{bool} \Rightarrow \text{bool}) \Rightarrow \text{bool} \Rightarrow \text{bool}$ has arity 2, then id , g id , g (g id) and so on are all values. Therefore, instead of looking directly at values we consider an extensional replacement.

Definition 13. Let \mathcal{B} be a set of data expressions closed under \triangleright . For $\iota \in \mathcal{S}$, let $\langle \iota \rangle_{\mathcal{B}} = \{d \in \mathcal{B} \mid \vdash d : \iota\}$. Inductively, let $\langle \sigma \times \tau \rangle_{\mathcal{B}} = \langle \sigma \rangle_{\mathcal{B}} \times \langle \tau \rangle_{\mathcal{B}}$ and $\langle \sigma \Rightarrow \tau \rangle_{\mathcal{B}} = \{A_{\sigma \Rightarrow \tau} \mid A \subseteq \langle \sigma \rangle_{\mathcal{B}} \times \langle \tau \rangle_{\mathcal{B}} \wedge \forall e \in \langle \sigma \rangle_{\mathcal{B}} \text{ there is at most one } u \text{ with } (e, u) \in A_{\sigma \Rightarrow \tau}\}$. We call the elements of any $\langle \sigma \rangle_{\mathcal{B}}$ deterministic extensional values.

Note that deterministic extensional values are data expressions in \mathcal{B} if σ is a sort, *pairs* if σ is a pair type, and sets of pairs labelled with a type otherwise; these sets are exactly partial functions, and can be used as such:

Definition 14. For $e \in \langle \sigma_1 \Rightarrow \dots \Rightarrow \sigma_n \Rightarrow \tau \rangle_{\mathcal{B}}$ and $u_1 \in \langle \sigma_1 \rangle_{\mathcal{B}}, \dots, u_n \in \langle \sigma_n \rangle_{\mathcal{B}}$, we inductively define $e(u_1, \dots, u_n) \subseteq \langle \tau \rangle_{\mathcal{B}}$:

- if $n = 0$, then $e(u_1, \dots, u_n) = e() = \{e\}$;
- if $n \geq 1$, then $e(u_1, \dots, u_n) = \bigcup_{A_{\sigma_n \Rightarrow \tau} \in e(u_1, \dots, u_{n-1})} \{o \in \langle \tau \rangle_{\mathcal{B}} \mid (u_n, o) \in A\}$.

By induction on n , each $e(u_1, \dots, u_n)$ has at most one element as would be expected of a partial function. We also consider a form of matching.

Definition 15. Fix a set \mathcal{B} of data expressions. An extensional expression has the form $\mathbf{f} \ e_1 \cdots e_n$ where $\mathbf{f} : \sigma_1 \Rightarrow \dots \Rightarrow \sigma_n \Rightarrow \tau \in \mathcal{D}$ and each $e_i \in \langle \sigma_i \rangle_{\mathcal{B}}$. Given a clause $\rho : \mathbf{f} \ \ell_1 \cdots \ell_k = r$ with $\mathbf{f} : \sigma_1 \Rightarrow \dots \Rightarrow \sigma_k \Rightarrow \tau \in \mathcal{F}$ and variable environment Γ , an ext-environment for ρ is a partial function η mapping each $x : \tau \in \Gamma$ to an element of $\langle \tau \rangle_{\mathcal{B}}$, such that $\ell_j \eta \in \langle \sigma_j \rangle_{\mathcal{B}}$ for $1 \leq j \leq n$. Here,

- $\ell\eta = \eta(\ell)$ if ℓ is a variable
- $\ell\eta = (\ell^{(1)}\eta, \ell^{(2)}\eta)$ if $\ell = (\ell^{(1)}, \ell^{(2)})$;
- $\ell\eta = \ell[x := \eta(x) \mid x \in \text{Var}(\ell)]$ otherwise (in this case, ℓ is a pattern with data order 0, so all its variables have data order 0, so each $\eta(x) \in \text{Data}$).

Then $\ell\eta$ is a deterministic extensional value for ℓ a pattern. We say ρ matches an extensional expression $\mathbf{f} \ e_1 \cdots e_k$ if there is an ext-environment η for ρ such that $\ell_i\eta = e_i$ for all $1 \leq i \leq k$. We call η the matching ext-environment.

Finally, for technical reasons we will need an ordering on extensional values:

Definition 16. We define a relation \sqsupseteq on extensional values of the same type:

- For $d, b \in \langle \iota \rangle_{\mathcal{B}}$ with $\iota \in \mathcal{S}$: $d \sqsupseteq b$ if $d = b$.
- For $(e_1, e_2), (u_1, u_2) \in \langle \sigma \times \tau \rangle_{\mathcal{B}}$: $(e_1, e_2) \sqsupseteq (u_1, u_2)$ if each $e_i \sqsupseteq u_i$.
- For $A_\sigma, B_\sigma \in \langle \sigma \rangle_{\mathcal{B}}$ with σ functional: $A_\sigma \sqsupseteq B_\sigma$ if for all $(e, u) \in B$ there is $u' \sqsupseteq u$ such that $(e, u') \in A$.

The algorithm. Let us now define our algorithm. We will present it in a general form—including a case **2d** which does not apply to deterministic programs—so we can reuse the algorithm in the non-deterministic settings to follow.

Algorithm 7 Let \mathbf{p} be a fixed, deterministic cons-free program, and suppose \mathbf{f}_1 has a type $\kappa_1 \Rightarrow \dots \Rightarrow \kappa_M \Rightarrow \kappa \in \mathcal{F}$.

Input: data expressions $d_1 : \kappa_1, \dots, d_M : \kappa_M$.

Output: The set of values b with $\llbracket \mathbf{p} \rrbracket(d_1, \dots, d_M) \mapsto b$.

1. Preparation.
 - (a) Let \mathbf{p}' be obtained from \mathbf{p} by the transformations of Lemma 1, and by adding a clause **start** $x_1 \cdots x_M = \mathbf{f}_1 \ x_1 \cdots x_M$ for a fresh symbol **start** (so that $\llbracket \mathbf{p} \rrbracket(d_1, \dots, d_M) \mapsto b$ iff $\mathbf{p}' \vdash^{\text{call}} \text{start } d_1 \cdots d_M \rightarrow b$).
 - (b) Denote $\mathcal{B} := \mathcal{B}_{d_1, \dots, d_M}^{\mathbf{p}}$ and let \mathcal{X} be the set of all “statements”:
 - i. $\vdash \mathbf{f} \ e_1 \cdots e_n \rightsquigarrow o$ for (a) $\mathbf{f} \in \mathcal{D}$ with $\mathbf{f} : \sigma_1 \Rightarrow \dots \Rightarrow \sigma_m \Rightarrow \kappa' \in \mathcal{F}$, (b) $0 \leq n \leq \text{arity}_{\mathbf{p}}(\mathbf{f})$ such that $\text{ord}(\sigma_{n+1} \Rightarrow \dots \Rightarrow \sigma_m \Rightarrow \kappa') \leq K$, (c) $e_i \in \langle \sigma_i \rangle_{\mathcal{B}}$ for $1 \leq i \leq n$ and (d) $o \in \langle \sigma_{n+1} \Rightarrow \dots \Rightarrow \sigma_m \Rightarrow \kappa' \rangle_{\mathcal{B}}$;
 - ii. $\eta \vdash t \rightsquigarrow o$ for (a) $\rho : \mathbf{f} \ \ell_1 \cdots \ell_k = s$ a clause in \mathbf{p}' , (b) $s \sqsupseteq t : \tau$, (c) $o \in \langle \tau \rangle_{\mathcal{B}}$ and (d) η an ext-environment for ρ .
 - (c) Mark statements of the form $\eta \vdash t \rightsquigarrow o$ in \mathcal{X} as confirmed if :
 - i. $t \in \mathcal{V}$ and $\eta(t) \sqsupseteq o$, or
 - ii. $t = \mathbf{c} \ t_1 \cdots t_m$ with $\mathbf{c} \in \mathcal{C}$ and $t\eta = o$.
 All statements not of either form are marked unconfirmed.
2. Iteration: repeat the following steps, until no further changes are made.
 - (a) For all unconfirmed statements $\vdash \mathbf{f} \ e_1 \cdots e_n \rightsquigarrow o$ in \mathcal{X} with $n < \text{arity}_{\mathbf{p}}(\mathbf{f})$: write $o = O_\sigma$ and mark the statement as confirmed if for all $(e_{n+1}, u) \in O$ there exists $u' \sqsupseteq u$ such that $\vdash \mathbf{f} \ e_1 \cdots e_{n+1} \rightsquigarrow u'$ is marked confirmed.
 - (b) For all unconfirmed statements $\vdash \mathbf{f} \ e_1 \cdots e_k \rightsquigarrow o$ in \mathcal{X} with $k = \text{arity}_{\mathbf{p}}(\mathbf{f})$:
 - i. find the first clause $\rho : \mathbf{f} \ \ell_1 \cdots \ell_k = s$ in \mathbf{p}' that matches $\mathbf{f} \ e_1 \cdots e_k$ and let η be the matching ext-environment (if any);

- ii. determine whether $\eta \vdash s \rightsquigarrow o$ is confirmed and if so, mark the statement $\mathbf{f} \ e_1 \cdots e_k \rightsquigarrow o$ as confirmed.
- (c) For all unconfirmed statements of the form $\eta \vdash \text{if } s_1 \text{ then } s_2 \text{ else } s_3 \rightsquigarrow o$ in \mathcal{X} , mark the statement confirmed if
 - i. both $\eta \vdash s_1 \rightsquigarrow \mathbf{true}$ and $\eta \vdash s_2 \rightsquigarrow o$ are confirmed, or
 - ii. both $\eta \vdash s_1 \rightsquigarrow \mathbf{false}$ and $\eta \vdash s_3 \rightsquigarrow o$ are confirmed.
- (d) For all unconfirmed statements $\eta \vdash \text{choose } s_1 \cdots s_n \rightsquigarrow o$ in \mathcal{X} , mark the statement as confirmed if $\eta \vdash s_i \rightsquigarrow o$ for any $i \in \{1, \dots, n\}$.
- (e) For all unconfirmed statements $\eta \vdash (s_1, s_2) \rightsquigarrow (o_1, o_2)$ in \mathcal{X} , mark the statement confirmed if both $\eta \vdash s_1 \rightsquigarrow o_1$ and $\eta \vdash s_2 \rightsquigarrow o_2$ are confirmed.
- (f) For all unconfirmed statements $\eta \vdash x \ s_1 \cdots s_n \rightsquigarrow o$ in \mathcal{X} with $x \in \mathcal{V}$, mark the statement as confirmed if there are $e_1 \in \langle \sigma_1 \rangle_{\mathcal{B}}, \dots, e_n \in \langle \sigma_n \rangle_{\mathcal{B}}$ such that each $\eta \vdash s_i \rightsquigarrow e_i$ is marked confirmed, and there exists $o' \in \eta(x)(e_1, \dots, e_n)$ such that $o' \sqsupseteq o$.
- (g) For all unconfirmed statements $\eta \vdash \mathbf{f} \ s_1 \cdots s_n \rightsquigarrow o$ in \mathcal{X} with $\mathbf{f} \in \mathcal{D}$, mark the statement as confirmed if there are $e_1 \in \langle \sigma_1 \rangle_{\mathcal{B}}, \dots, e_n \in \langle \sigma_n \rangle_{\mathcal{B}}$ such that each $\eta \vdash s_i \rightsquigarrow e_i$ is marked confirmed, and:
 - i. $n \leq \text{arity}_{\mathbf{p}}(\mathbf{f})$ and $\vdash \mathbf{f} \ e_1 \cdots e_n \rightsquigarrow o$ is marked confirmed, or
 - ii. $n > k := \text{arity}_{\mathbf{p}}(\mathbf{f})$ and there are u, o' such that $\vdash \mathbf{f} \ e_1 \cdots e_k \rightsquigarrow u$ is marked confirmed and $u(e_{k+1}, \dots, e_n) \ni o' \sqsupseteq o$.
- 3. Completion: return $\{b \mid b \in \mathcal{B} \wedge \vdash \text{start } d_1 \cdots d_M \rightsquigarrow b \text{ is marked confirmed}\}$.

Note that, for programs of data order 0, this algorithm closely follows the earlier sketch. Values of a higher type are abstracted to deterministic extensional values. The use of \sqsupseteq is needed because a value of higher type is associated to many extensional values; e.g., to confirm a statement $\vdash \text{plus } 3 \rightsquigarrow \{(1, 4), (0, 3)\}_{\text{nat} \Rightarrow \text{nat}}$ in some program, it may be necessary to first confirm $\vdash \text{plus } 3 \rightsquigarrow \{(0, 3)\}_{\text{nat} \Rightarrow \text{nat}}$.

The complexity of the algorithm relies on the following key observation:

Lemma 8. *Let \mathbf{p} be a cons-free program of data order K . Let Σ be the set of all types σ with $\text{ord}(\sigma) \leq K$ which occur as part of an argument type, or as an output type of some $\mathbf{f} \in \mathcal{D}$. Suppose that, given input of total size n , $\langle \sigma \rangle_{\mathcal{B}}$ has cardinality at most $F(n)$ for all $\sigma \in \Sigma$, and testing whether $e_1 \sqsupseteq e_2$ for $e_1, e_2 \in \llbracket \sigma \rrbracket_{\mathcal{B}}$ takes at most $F(n)$ steps. Then Algorithm 7 runs in $\text{TIME}(a \cdot F(n)^b)$ for some a, b .*

Here, the cardinality $\text{Card}(A)$ of a set A is just the number of elements of A .

Proof (Sketch). Due to the use of \mathbf{p}' , all intensional values occurring in Algorithm 7 are in $\bigcup_{\sigma \in \Sigma} \langle \sigma \rangle_{\mathcal{B}}$. Writing \mathbf{a} for the greatest number of arguments any defined symbol \mathbf{f} or variable x in \mathbf{p}' may take and \mathbf{r} for the greatest number of sub-expressions of any right-hand side in \mathbf{p}' (which is independent of the input!), \mathcal{X} contains at most $\mathbf{a} \cdot |\mathcal{D}| \cdot F(n)^{\mathbf{a}+1} + |\mathbf{p}'| \cdot \mathbf{r} \cdot F(n)^{\mathbf{a}+1}$ statements. Since in all but the last step of the iteration at least one statement is flipped from unconfirmed to confirmed, there are at most $|\mathcal{X}| + 1$ iterations, each considering $|\mathcal{X}|$ statements. It is easy to see that the individual steps in both the preparation and iteration are all polynomial in $|\mathcal{X}|$ and $F(n)$, resulting in a polynomial overall complexity.

(See Appendix D for the complete proof.) \square

The result follows as $\text{Card}(\langle\sigma\rangle_{\mathcal{B}})$ is given by a tower of exponentials in $\text{ord}(\sigma)$:

Lemma 9. *If $1 \leq \text{Card}(\mathcal{B}) < N$, then for each σ of length L (where the length of a type is the number of sorts occurring in it, including repetitions), with $\text{ord}(\sigma) \leq K$: $\text{Card}(\langle\sigma\rangle_{\mathcal{B}}) < \exp_2^K(N^L)$. Testing $e \sqsubseteq u$ for $e, u \in \langle\sigma\rangle_{\mathcal{B}}$ takes at most $\exp_2^K(N^{(L+1)^3})$ comparisons between elements of \mathcal{B} .*

Proof (Sketch). An easy induction on the form of σ , using that $\exp_2^K(X) \cdot \exp_2^K(Y) \leq \exp_2^K(X \cdot Y)$ for $X \geq 2$, and that for $A_{\sigma_1 \Rightarrow \sigma_2}$, each key $e \in \langle\sigma_1\rangle_{\mathcal{B}}$ is assigned one of $\text{Card}(\langle\sigma_2\rangle_{\mathcal{B}}) + 1$ choices: an element u of $\langle\sigma_2\rangle_{\mathcal{B}}$ such that $(e, u) \in A$, or non-membership. The second part (regarding \sqsubseteq) uses the first.

(See Appendix D for the complete proof.) \square

We will postpone showing correctness of the algorithm until Section 6.3, where we can show the result together with the one for non-deterministic programs. Assuming correctness for now, we may conclude:

Lemma 10. *Every decision problem accepted by a deterministic cons-free program p with data order K is in EXP^KTIME .*

Proof. We will see in Lemma 20 in Section 6.3 that $\llbracket p \rrbracket(d_1, \dots, d_M) \mapsto b$ if and only if Algorithm 7 returns the set $\{b\}$. For a program of data order K , Lemmas 8 and 9 together give that Algorithm 7 operates in $\text{TIME}(\exp_2^K(n))$. \square

Theorem 1. *The class of deterministic cons-free programs with data order K characterises EXP^KTIME for all $K \in \mathbb{N}$.*

Proof. A combination of Lemmas 6 and 10. \square

6 Non-deterministic characterisations

A natural question is what happens if we do not limit interest to deterministic programs. For data order 0, Bonfante [4] shows that adding the choice operator to Jones' language does not increase expressivity. We will recover this result for our generalised language in Section 7. However, in the higher-order setting, non-deterministic choice *does* increase expressivity—dramatically so. We have:

	data order 0	data order 1	data order 2	data order 3	...
cons-free	P	ELEMENTARY	ELEMENTARY	ELEMENTARY	...

As before, we will show the result—for data orders 1 and above—in two parts: in Section 6.1 we see that cons-free programs of data order 1 suffice to accept all problems in ELEMENTARY; in Section 6.2 we see that they cannot go beyond.

6.1 Simulating TMs using (non-deterministic) cons-free programs

We start by showing how Turing Machines in **ELEMENTARY** can be simulated by non-deterministic cons-free programs. For this, we reuse the core simulation from Figure 7. The reason for the jump in expressivity lies in Lemma 3: by taking advantage of non-determinism, we can count up to arbitrarily high numbers.

Lemma 11. *If there is a P -counting module C_π with data order $K \leq 1$, there is a (non-deterministic) $(\lambda n. 2^{P(n)-1})$ -counting module $C_{\psi[\pi]}$ with data order 1.*

Proof. We let $\alpha_{\psi[\pi]} := \text{bool} \Rightarrow \alpha_\pi$ (which has type order $\max(1, \text{ord}(\alpha_\pi))$), and:

- $\mathcal{A}_{\psi[\pi]}^n :=$ the set of those values $v : \alpha_{\psi[\pi]}$ such that:
 - there is $w \in \mathcal{A}_\pi$ with $\langle w \rangle_\pi^n = 0$ such that $\mathbf{p}_{\psi[\pi]} \vdash^{\text{call}} v \text{ true} \rightarrow w$;
 - there is $w \in \mathcal{A}_\pi$ with $\langle w \rangle_\pi^n = 0$ such that $\mathbf{p}_{\psi[\pi]} \vdash^{\text{call}} v \text{ false} \rightarrow w$;
 and for all $1 \leq i < P(n)$ exactly one of the following holds:
 - there is $w \in \mathcal{A}_\pi^n$ with $\langle w \rangle_\pi^n = i$ such that $\mathbf{p}_{\psi[\pi]} \vdash^{\text{call}} v \text{ true} \rightarrow w$;
 - there is $w \in \mathcal{A}_\pi^n$ with $\langle w \rangle_\pi^n = i$ such that $\mathbf{p}_{\psi[\pi]} \vdash^{\text{call}} v \text{ false} \rightarrow w$;
 We will say that $v \text{ true} \mapsto i$ or $v \text{ false} \mapsto i$ respectively.
- $\langle v \rangle_{\psi[\pi]}^n := \sum_{i=1}^{P(n)-1} \{2^{P(n)-1-i} \mid v \text{ true} \mapsto i\}$;
- $\mathbf{p}_{\psi[\pi]}$ be given by Figure 9 appended to \mathbf{p}_π , and $\mathcal{D}_{\psi[\pi]}$ by the symbols in $\mathbf{p}_{\psi[\pi]}$.

So, we interpret a value v as the number given by the bitstring $b_1 \dots b_{P(n)-1}$ (most significant digit first), where b_i is 1 if $v \text{ true}$ evaluates to a value representing i in C_π , and b_i is 0 otherwise—so exactly if $v \text{ false}$ evaluates to such a value. \square

To understand the counting program, consider 4, with bit representation 100. If 0, 1, 2, 3 are represented in C_π by values O, w_1, w_2, w_3 respectively, then in $C_{\psi[\pi]}$, the number 4 corresponds for example to Q :

$$\text{st1 } w_1 (\text{st0 } w_2 (\text{st0 } w_3 (\text{base}_{\psi[\pi]} O)))$$

The null-value O functions as a default, and is a possible value of both $Q \text{ true}$ and $Q \text{ false}$ for any function Q representing a bitstring.

The non-determinism comes into play when determining whether $Q \text{ true} \mapsto i$ or not: we can evaluate $F \text{ true}$ to *some* value, but this may not be the value we need. Therefore, we find some value of both $F \text{ true}$ and $F \text{ false}$; if either represents i in C_π , then we have confirmed or rejected that $b_i = 1$. If both evaluations give a different value, we repeat the test. This gives a non-terminating program, but there is always exactly one value b such that $\mathbf{p}_{\psi[\pi]} \vdash^{\text{call}} \text{bitset}_{\psi[\pi]} \text{ cs } F \text{ } i \rightarrow b$.

The $\text{seed}_{\psi[\pi]}$ function generates the bit string $1 \dots 1$, so the function F with $F \text{ true} \mapsto i$ for all $i \in \{0, \dots, P(n) - 1\}$ and $F \text{ false} \mapsto i$ for only $i = 0$. The $\text{zero}_{\psi[\pi]}$ function iterates through $b_{P(n)-1}, b_{P(n)-2}, \dots, b_1$ and tests whether all bits are set to 0. The clauses for $\text{pred}_{\psi[\pi]}$ assume given a bitstring $b_1 \dots b_{i-1} 10 \dots 0$, and recursively build $b_1 \dots b_{i-1} 01 \dots 1$ in the parameter G .

– core elements; **sti** n F sets bit n in F to the value i

```

baseψ[π]  $x$   $b = x$ 
st1ψ[π]  $n$   $F$  true = choose  $n$  ( $F$  true)      st0ψ[π]  $n$   $F$  true =  $F$  true
st1ψ[π]  $n$   $F$  false =  $F$  false                st0ψ[π]  $n$   $F$  false = choose  $n$  ( $F$  false)

```

– testing bit values (using non-determinism and non-termination)

```

bitsetψ[π]  $cs$   $F$   $i$  = if equalπ  $cs$  ( $F$  true)  $i$  then true
                      else if equalπ  $cs$  ( $F$  false)  $i$  then false
                      else bitsetψ[π]  $cs$   $F$   $i$ 

```

– the seed function

```

nulπ  $cs$  = nul'π  $cs$  (seedπ  $cs$ )
nul'π  $cs$   $n$  = if zeroπ  $cs$   $n$  then  $n$  else nul'π  $cs$  (predπ  $cs$   $n$ )
seedψ[π]  $cs$  = seed'ψ[π]  $cs$  (seedπ  $cs$ ) (baseψ[π] (nulπ  $cs$ ))
seed'ψ[π]  $cs$   $i$   $F$  = if zeroπ  $cs$   $i$  then  $F$  else seed'ψ[π]  $cs$  (predπ  $cs$   $i$ ) (st1ψ[π]  $i$   $F$ )

```

– the zero test

```

zeroψ[π]  $cs$   $F$  = zero'ψ[π]  $cs$   $F$  (seedπ  $cs$ )
zero'ψ[π]  $cs$   $F$   $i$  = if zeroπ  $i$  then true
                  else if bitsetψ[π]  $cs$   $F$   $i$  then false
                  else zero'ψ[π]  $cs$   $F$  (predπ  $cs$   $i$ )

```

– the predecessor

```

predψ[π]  $cs$   $F$  = prψ[π]  $cs$   $F$  (seedπ  $cs$ ) (baseψ[π] (nulπ  $cs$ ))
prψ[π]  $cs$   $F$   $i$   $G$  = if bitsetψ[π]  $cs$   $F$   $i$  then cpψ[π]  $cs$   $F$  (predπ  $cs$   $i$ ) (st0ψ[π]  $i$   $G$ )
                  else prψ[π]  $cs$   $F$  (predπ  $cs$   $i$ ) (st1ψ[π]  $i$   $G$ )
cp  $cs$   $F$   $i$   $G$  = if zeroπ  $cs$   $i$  then  $G$ 
                  else if bitsetψ[π]  $cs$   $F$   $i$  then cpψ[π]  $cs$   $F$  (predπ  $cs$   $i$ ) (st1ψ[π]  $i$   $G$ )
                  else cpψ[π]  $cs$   $F$  (predπ  $cs$   $i$ ) (st0ψ[π]  $i$   $G$ )

```

Fig. 9. Clauses for the counting module $C_{\psi[\pi]}$.

Example 10. Consider an input string of length 3, say **false::false::true::[]**. Recall from Lemma 4 that there is a $(\lambda n. n + 1)$ -counting module $C_{\langle 1,1 \rangle}$ representing $i \in \{0, \dots, 3\}$ as suffixes of length i from the input string. Therefore, there is also a second-order $(\lambda n. 2^n)$ -counting module $C_{\psi[\langle 1,1 \rangle]}$ representing $i \in \{0, \dots, 7\}$. The number 6—with bitstring 110—is represented by the value w_6 :

$$w_6 = \mathbf{st1}_{\psi[\langle 1,1 \rangle]} (\mathbf{true}::[]) (\mathbf{st1}_{\psi[\langle 1,1 \rangle]} (\mathbf{false}::\mathbf{true}::[]) (\mathbf{st0}_{\psi[\langle 1,1 \rangle]} (\mathbf{false}::\mathbf{false}::\mathbf{true}::[]) (\mathbf{cons}_{\psi[\langle 1,1 \rangle]} [])) : \mathbf{bool} \Rightarrow \mathbf{list}$$

But then there is also a $(\lambda n. 2^{2^n} - 1)$ -counting module $C_{\psi[\psi[\langle 1,1 \rangle]]}$, representing $i \in \{0, \dots, 2^7 - 1\}$. For example 97—with bit vector 1100001—is represented by:

$$S = \mathbf{st1}_{\psi[\psi[\langle 1,1 \rangle]]} w_1 (\mathbf{st1}_{\psi[\psi[\langle 1,1 \rangle]]} w_2 (\mathbf{st0}_{\psi[\psi[\langle 1,1 \rangle]]} w_3 (\mathbf{st0}_{\psi[\psi[\langle 1,1 \rangle]]} w_4 (\mathbf{st0}_{\psi[\psi[\langle 1,1 \rangle]]} w_5 (\mathbf{st0}_{\psi[\psi[\langle 1,1 \rangle]]} w_6 (\mathbf{st1}_{\psi[\psi[\langle 1,1 \rangle]]} w_7 (\mathbf{cons}_{\psi[\psi[\langle 1,1 \rangle]]} w_7)))))$$

Here $\mathbf{st1}_{\psi[\psi[\langle 1,1 \rangle]]}$ and $\mathbf{st0}_{\psi[\psi[\langle 1,1 \rangle]]}$ have the type $(\mathbf{bool} \Rightarrow \mathbf{list}) \Rightarrow (\mathbf{bool} \Rightarrow \mathbf{bool} \Rightarrow \mathbf{list}) \Rightarrow \mathbf{bool} \Rightarrow \mathbf{bool} \Rightarrow \mathbf{list}$ and each w_i represents i in $C_{\psi[\langle 1,1 \rangle]}$, as shown for w_6 above. Note: S **true** $\mapsto w_1, w_2, w_7$ and S **false** $\mapsto w_3, w_4, w_5, w_6$.

Since $2^{2^m} - 1 \geq 2^m$ for all $m \geq 2$, we can count up to arbitrarily high bounds using this module. Thus, already with data order 1, we can simulate Turing Machines operating in TIME $(\exp_2^K(n))$ for any K .

Lemma 12. *Every decision problem in ELEMENTARY is accepted by a non-deterministic cons-free program with data order 1.*

Proof. A decision problem is in ELEMENTARY if it is in some $\text{EXP}^K \text{TIME}$ which, by Lemma 3, is certainly the case if for any a, b there is a Q -counting module with $Q \geq \lambda n. \exp_2^K(a \cdot n^b)$. Such a module exists for data order 1 by Lemma 11. \square

6.2 Simulating cons-free programs using an algorithm

Towards a characterisation, we must also see that every decision problem accepted by a cons-free program is in ELEMENTARY—so that the result of every such program can be found by an algorithm operating in $\text{TIME}(\exp_2^K(a \cdot n^b))$ for some a, b, K . We can reuse Algorithm 7 by altering the definition of $\langle \sigma \rangle_{\mathcal{B}}$.

Definition 17. *Let \mathcal{B} be a set of data expressions closed under \triangleright . For $\iota \in \mathcal{S}$, let $\llbracket \iota \rrbracket_{\mathcal{B}} = \{d \in \mathcal{B} \mid \vdash d : \iota\}$. Inductively, define $\llbracket \sigma \times \tau \rrbracket_{\mathcal{B}} = \llbracket \sigma \rrbracket_{\mathcal{B}} \times \llbracket \tau \rrbracket_{\mathcal{B}}$ and $\llbracket \sigma \Rightarrow \tau \rrbracket_{\mathcal{B}} = \{A_{\sigma \Rightarrow \tau} \mid A \subseteq \llbracket \sigma \rrbracket_{\mathcal{B}} \times \llbracket \tau \rrbracket_{\mathcal{B}}\}$. We call the elements of any $\llbracket \sigma \rrbracket_{\mathcal{B}}$ non-deterministic extensional values.*

Where the elements of $\langle \sigma \Rightarrow \tau \rangle_{\mathcal{B}}$ are partial functions, $\llbracket \sigma \Rightarrow \tau \rrbracket_{\mathcal{B}}$ contains arbitrary relations: a value v is associated to a set of pairs (e, u) such that v *might* evaluate to u . The notions of extensional expression, $e(u_1, \dots, u_n)$ and \sqsubseteq immediately extend to non-deterministic extensional values. Thus we can define:

Algorithm 13 *Let \mathbf{p} be a fixed, non-deterministic cons-free program, with $\mathbf{f}_1 : \kappa_1 \Rightarrow \dots \Rightarrow \kappa_M \Rightarrow \kappa \in \mathcal{F}$.*

Input: *data expressions $d_1 : \kappa_1, \dots, d_M : \kappa_M$.*

Output: *The set of values b with $\llbracket \mathbf{p} \rrbracket(d_1, \dots, d_M) \mapsto b$.*

Execute Algorithm 7, but using $\llbracket \sigma \rrbracket_{\mathcal{B}}$ in place of $\langle \sigma \rangle_{\mathcal{B}}$.

In Section 6.3, we will see that indeed $\llbracket \mathbf{p} \rrbracket(d_1, \dots, d_M) \mapsto b$ if and only if Algorithm 13 returns a set containing b . But as before, we first consider complexity. To properly analyse this, we introduce the new notion of *arrow depth*.

Definition 18. *A type's arrow depth is given by: $\text{depth}(\iota) = 0$, $\text{depth}(\sigma \times \tau) = \max(\text{depth}(\sigma), \text{depth}(\tau))$ and $\text{depth}(\sigma \Rightarrow \tau) = 1 + \max(\text{depth}(\sigma), \text{depth}(\tau))$.*

Now the cardinality of each $\llbracket \sigma \rrbracket_{\mathcal{B}}$ can be expressed using its arrow depth:

Lemma 14. *If $1 \leq \text{Card}(\mathcal{B}) < N$, then for each σ of length L , with $\text{depth}(\sigma) \leq K$: $\text{Card}(\llbracket \sigma \rrbracket_{\mathcal{B}}) < \exp_2^K(N^L)$. Testing $e \sqsubseteq u$ for $e, u \in \llbracket \sigma \rrbracket_{\mathcal{B}}$ takes at most $\exp_2^K(N^{(L+1)^3})$ comparisons.*

Proof (Sketch). A straightforward induction on the form of σ , like Lemma 9.

(See Appendix D for the complete proof.) \square

Thus, once more assuming correctness for now, we may conclude:

Lemma 15. *Every decision problem accepted by a non-deterministic cons-free program p is in ELEMENTARY.*

Proof. We will see in Lemma 18 in Section 6.3 that $\llbracket p \rrbracket(d_1, \dots, d_M) \mapsto b$ if and only if Algorithm 13 returns a set containing b . Since all types have an arrow depth and the set Σ in Lemma 8 is finite, Algorithm 13 operates in some $\text{TIME}(\exp_2^K(n))$. Thus, the problem is in $\text{EXP}^K\text{TIME} \subseteq \text{ELEMENTARY}$. \square

Theorem 2. *The class of non-deterministic cons-free programs with data order K characterises ELEMENTARY for all $K \in \mathbb{N} \setminus \{0\}$.*

Proof. A combination of Lemmas 12 and 15. \square

6.3 Correctness proofs of Algorithms 7 and 13

Algorithms 7 and 13 are the same—merely parametrised with a different set of extensional values to be used in step 1b. Due to this similarity, and because $\langle \sigma \rangle_{\mathcal{B}} \subseteq \llbracket \sigma \rrbracket_{\mathcal{B}}$, we can largely combine their correctness proofs. The proofs are somewhat intricate, however; all details are provided in Appendix E.

We begin with *soundness*:

Lemma 16. *If Algorithm 7 or 13 returns a set $A \cup \{b\}$, then $\llbracket p \rrbracket(d_1, \dots, d_M) \mapsto b$.*

Proof (Sketch). We define for every value $v : \sigma$ and $e \in \llbracket \sigma \rrbracket_{\mathcal{B}}$: $v \Downarrow e$ iff: (a) $\sigma \in \mathcal{S}$ and $v = e$; or (b) $\sigma = \sigma_1 \times \sigma_2$ and $v = (v_1, v_2)$ and $e = (e_1, e_2)$ with $v_1 \Downarrow e_1$ and $v_2 \Downarrow e_2$; or (c) $\sigma = \sigma_1 \Rightarrow \sigma_2$ and $e = A_\sigma$ with $A \subseteq \{(u_1, u_2) \mid u_1 \in \llbracket \sigma_1 \rrbracket_{\mathcal{B}} \wedge u_2 \in \llbracket \sigma_2 \rrbracket_{\mathcal{B}} \wedge \text{for all values } w_1 : \sigma_1 \text{ with } w_1 \Downarrow u_1 \text{ there is some value } w_2 : \sigma_2 \text{ with } w_2 \Downarrow u_2 \text{ such that } p' \vdash^{\text{call}} v \ w_1 \rightarrow w_2\}$.

We now prove two statements together by induction on the confirmation time in Algorithm 7, which we consider equipped with *unspecified* subsets $[\sigma]$ of $\llbracket \sigma \rrbracket_{\mathcal{B}}$:

1. Let: (a) $f : \sigma_1 \Rightarrow \dots \Rightarrow \sigma_m \Rightarrow \kappa \in \mathcal{F}$ be a defined symbol; (b) $v_1 : \sigma_1, \dots, v_n : \sigma_n$ be values, for $1 \leq n \leq \text{arity}_p(f)$; (c) $e_1 \in \llbracket \sigma_1 \rrbracket_{\mathcal{B}}, \dots, e_n \in \llbracket \sigma_n \rrbracket_{\mathcal{B}}$ be such that each $v_i \Downarrow e_i$; (d) $o \in \llbracket \sigma_{n+1} \Rightarrow \dots \Rightarrow \sigma_m \Rightarrow \kappa \rrbracket_{\mathcal{B}}$. If $\vdash f \ e_1 \dots e_n \rightsquigarrow o$ is eventually confirmed, then $p' \vdash^{\text{call}} f \ v_1 \dots v_n \rightarrow w$ for some w with $w \Downarrow o$.
2. Let: (a) $\rho : f \ \ell = s$ be a clause in p' ; (b) $t : \tau$ be a sub-expression of s ; (c) η be an ext-environment for ρ ; (d) γ be an environment such that $\gamma(x) \Downarrow \eta(x)$ for all $x \in \text{Var}(f \ \ell)$; (e) $o \in \llbracket \tau \rrbracket_{\mathcal{B}}$. If the statement $\eta \vdash t \rightsquigarrow o$ is eventually confirmed, then $p', \gamma \vdash t \rightarrow w$ for some w with $w \Downarrow o$.

Given the way p' is defined from p , the lemma follows from the first statement. The induction is easy, but requires minor sub-steps such as transitivity of \Downarrow . \square

The harder part, where the algorithms diverge, is *completeness*:

Lemma 17. *If $\llbracket p \rrbracket(d_1, \dots, d_M) \mapsto b$, then Algorithm 13 returns a set $A \cup \{b\}$.*

Proof (Sketch). If $\llbracket p \rrbracket(d_1, \dots, d_M) \mapsto b$, then $p' \vdash^{\text{call}} \text{start } d_1 \dots d_M \rightarrow b$. We label the nodes in the derivation trees with strings of numbers (a node with label l has immediate subtrees of the form $l \cdot i$), and let $>$ denote lexicographic comparison of these strings, and \succ lexicographic comparison without prefixes (e.g., $1 \cdot 2 > 1$ but not $1 \cdot 2 \succ 1$). We define the following function:

- $\psi(v, l) = v$ if $v \in \mathcal{B}$, and $\psi((v_1, v_2), l) = (\psi(v_1, l), \psi(v_2, l))$;
- for $\mathbf{f} \ v_1 \dots v_n : \tau = \sigma_{n+1} \Rightarrow \dots \Rightarrow \sigma_m \Rightarrow \kappa$ with $m > n$, let $\psi(\mathbf{f} \ v_1 \dots v_n, l) = \{(e_{n+1}, u) \mid \exists q \succ p > l \text{ [the subtree with index } p \text{ has a root } p' \vdash^{\text{call}} \mathbf{f} \ v_1 \dots v_{n+1} \rightarrow w \text{ with } \psi(w, q) = u \text{ and } e_{n+1} \sqsubseteq' \psi(v_{n+1}, p)]\}_\tau$.

Here, \sqsubseteq' is defined the same as \sqsubseteq , except that $A_\sigma \sqsubseteq' B_\sigma$ iff $A \sqsupseteq B$. Note that clearly $A \sqsubseteq' B$ implies $A \sqsubseteq B$, and that \sqsubseteq' is transitive by transitivity of \sqsubseteq . Then, using induction on the labels of the tree in reverse lexicographical order (so going through the tree right-to-left, top-to-bottom), we can prove:

1. If the subtree labelled l has root $p' \vdash^{\text{call}} \mathbf{f} \ v_1 \dots v_n \rightarrow w$, then for all e_1, \dots, e_n such that each $e_i \sqsubseteq' \psi(v_i, l)$, and for all $p \succ l$ there exists $o \sqsubseteq' \psi(w, p)$ such that $\vdash \mathbf{f} \ e_1 \dots e_n \leadsto o$ is eventually confirmed.
2. If the subtree labelled l has root p' , $\gamma \vdash t \rightarrow w$ and $\eta(x) \sqsubseteq' \psi(\gamma(x), l)$ for all $x \in \text{Var}(t)$, then for all $p \succ l$ there exists $o \sqsubseteq' \psi(w, p)$ such that $\eta \Vdash t \leadsto o$ is eventually confirmed.

Assigning the main tree a label 0 (to secure that $p \succ 0$ exists), we obtain that $\vdash \text{start } d_1 \dots d_M \leadsto b$ is eventually confirmed, so b is indeed returned. \square

By Lemmas 16 and 17 together we may immediately conclude:

Lemma 18. $\llbracket p \rrbracket(d_1, \dots, d_M) \mapsto b$ iff Algorithm 13 returns a set containing b .

The proof of the general case provides a basis for the deterministic case:

Lemma 19. If $\llbracket p \rrbracket(d_1, \dots, d_M) \mapsto b$ and p is deterministic, then Algorithm 7 returns a set $A \cup \{b\}$.

Proof (Sketch). We define a consistency measure \wr on non-deterministic extensional values: $e \wr u$ iff $e = u \in \mathcal{B}$, or $e = (e_1, e_2)$, $u = (u_1, u_2)$, $e_1 \wr u_1$ and $e_2 \wr u_2$, or $e = A_\sigma$, $u = B_\sigma$ and for all $(e_1, u_1) \in A$ and $(e_2, u_2) \in B$: $e_1 \wr e_2$ implies $u_1 \wr u_2$.

In the proof of Lemma 17, we trace a derivation in the algorithm. In a deterministic program, we can see that if both $\vdash \mathbf{f} \ e_1 \dots e_n \rightarrow o$ and $\vdash \mathbf{f} \ e'_1 \dots e'_n \rightarrow o'$ are confirmed, and each $e_i \wr e'_i$, then $o \wr o'$ —and similar for statements $\eta \vdash s \Rightarrow o$. We use this to remove statements which are not necessary, ultimately leaving only those which use deterministic extensional values as used in Algorithm 7. \square

Lemma 20. $\llbracket p \rrbracket(d_1, \dots, d_M) \mapsto b$ iff Algorithm 7 returns a set containing b .

Proof. This is a combination of Lemmas 16 and 19. \square

Note that it is a priori not clear that Algorithm 7 returns only one value; however, this is obtained as a consequence of Lemma 20.

7 Recovering the EXPTIME hierarchy

While interesting, Lemma 12 exposes a problem: non-determinism is unexpectedly powerful in the higher-order setting. If we still want to use non-deterministic programs towards characterising non-deterministic complexity classes, we must surely start by considering restrictions which avoid this explosion of expressivity.

One direction is to consider *arrow depth* instead of data order. Using Lemma 14, we easily recover the original hierarchy—and obtain the last line of Figure 1.

	arrow depth 0	arrow depth 1	arrow depth 2	...
cons-free	$P = \text{EXP}^0\text{TIME}$	$\text{EXP} = \text{EXP}^1\text{TIME}$	EXP^2TIME	...

Theorem 3. *The class of non-deterministic cons-free programs where all variables are typed with a type of arrow depth K characterises EXP^KTIME .*

Proof (Sketch). Both in the base program in Figure 7, and in the counting modules of Lemmas 4 and 5, type order and arrow depth coincide. Thus every decision problem in EXP^KTIME is accepted by a cons-free program with “data arrow depth” K . For the other direction, the proof of Lemma 1 is trivially adapted to use arrow depth rather than type order. Thus, altering the preparation step in Algorithm 13 gives an algorithm which determines the possible outputs of a program with data arrow depth K , with the desired complexity by Lemma 14. \square

A downside is that, by moving away from data order, this result is hard to compare with other characterisations using cons-free programs. An alternative is to impose a restriction alongside cons-freeness: *unitary variables*. This gives no restrictions in the setting with data order 0—thus providing the first column in the table from Section 6—and brings us the second-last line in Figure 1:

	data order 0	data order 1	data order 2	data order 3
cons-free unitary variables	$P = \text{EXP}^0\text{TIME}$	$\text{EXP} = \text{EXP}^1\text{TIME}$	EXP^2TIME	EXP^3TIME

Definition 19. *A program p has unitary variables if clauses are typed with an assignment mapping each variable x to a type κ or $\sigma \Rightarrow \kappa$, with $\text{ord}(\kappa) = 0$.*

Thus, in a program with unitary variables, a variable of a type $(\text{list} \times \text{list} \times \text{list}) \Rightarrow \text{list}$ is admitted, but $\text{list} \Rightarrow \text{list} \Rightarrow \text{list} \Rightarrow \text{list}$ is not. The crucial difference is that the former must be applied to all its arguments at the same time, while the latter may be partially applied. This avoids the problem of Lemma 11.

Theorem 4. *The class of (deterministic or non-deterministic) cons-free programs with unitary variables of data order K characterises EXP^KTIME .*

Proof (Sketch). Both the base program in Figure 7 and the counting modules of Lemmas 4 and 5 have unitary variables, and are deterministic—this gives one direction. For the other, let a recursively unitary type be κ or $\sigma \Rightarrow \kappa$ with $\text{ord}(\kappa) = 0$ and σ recursively unitary. The transformations of Lemma 1 are easily extended to transform a program with unitary variables of type order $\leq K$ to one where all (sub-)expressions have a recursively unitary type. Since here data order and arrow depth are the same in this case, we complete with Theorem 3. \square

8 Conclusion and future work

We have studied the effect of combining higher types and non-determinism for cons-free programs. This has resulted in the—highly surprising—conclusion that naively adding non-deterministic choice to a language that characterises the EXP^KTIME hierarchy for increasing data orders immediately increases the expressivity of the language to **ELEMENTARY**. Recovering a more fine-grained complexity hierarchy can be done, but at the cost of further syntactical restrictions.

The primary goal that we will pursue in future work is to use non-deterministic cons-free programs to characterise hierarchies of *non*-deterministic complexity classes such as NEXP^KTIME for $K \in \mathbb{N}$. In addition, it would be worthwhile to make a full study of the ramifications of imposing restrictions on recursion, such as tail-recursion or primitive recursion, in combination with non-determinism and higher types (akin to the study of primitive recursion in a successor-free language done in [15]). We also intend to study characterisations of classes more restrictive than **P**, such as **LOGTIME** and **LOGSPACE**.

Finally, given the surprising nature of our results, we urge readers to investigate the effect of adding non-determinism to other programming languages used in implicit complexity that manipulate higher-order data. We conjecture that the effect on expressivity there will essentially be the same as what we have observed.

References

1. S. Bellantoni. PhD thesis, University of Toronto, 1993.
2. S. Bellantoni and S. Cook. A new recursion-theoretic characterization of the polytime functions. *Computational Complexity*, 2:97–110, 1992.
3. A. Ben-Amram and H. Petersen. CONS-free programs with tree input (extended abstract). In *ICALP*, volume 1443 of *LNCS*, pages 271–282, 1998.
4. G. Bonfante. Some programming languages for logspace and ptime. In *AMAST*, volume 4019 of *LNCS*, pages 66–80, 2006.
5. P. Clote. Computation models and function algebras. In *Handbook of Computability Theory*, pages 589–681. Elsevier, 1999.
6. S.A. Cook. Characterizations of pushdown machines in terms of time-bounded computers. *journal of the ACM*, 18(1):4–18, 1971.
7. D. de Carvalho and J. Simonsen. An implicit characterization of the polynomial-time decidable sets by cons-free rewriting. In *RTA-TLCA*, volume 8560 of *LNCS*, pages 179–193, 2014.
8. A. Goerdt. Characterizing complexity classes by general recursive definitions in higher types. *Information and Computation*, 101(2):202–218, 1992.
9. A. Goerdt. Characterizing complexity classes by higher type primitive recursive definitions. *Theoretical Computer Science*, 100(1):45–66, 1992.
10. N. Immerman. *Descriptive Complexity*. Springer-Verlag, 1999.
11. N. Jones. *Computability and Complexity from a Programming Perspective*. MIT Press, 1997.
12. N. Jones. The expressive power of higher-order types or, life without CONS. *Journal of Functional Programming*, 11(1):55–94, 2001.
13. A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. An analysis of ML typability. *Journal of the ACM*, 41(2):368–398, 1994.

14. C. Kop and J. Simonsen. Complexity hierarchies and higher-order cons-free rewriting. In *FSCD*, volume 52 of *LIPICs*, pages 23:1–23:18, 2016.
15. L. Kristiansen and B.M.W. Mender. Non-determinism in gödel’s system T. *Theory of Computing Systems*, 51(1):85–105, 2012.
16. L. Kristiansen and K.-H. Niggl. Implicit computational complexity on the computational complexity of imperative programming languages. *Theoretical Computer Science*, 318(1):139 – 161, 2004.
17. L. Kristiansen and P.J. Voda. Programming languages capturing complexity classes. *Nordic Journal of Computing*, 12(2):89–115, 2005.
18. U. Dal Lago. A short introduction to implicit computational complexity. In *Lectures on Logic and Computation: ESSLLI 2010/2011*, pages 89–109. 2012.
19. I. Oitavem. A recursion-theoretic approach to NP. *Annals of Pure and Applied Logic*, 162(8):661–666, 2011.
20. C. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.
21. M. Sipser. *Introduction to the Theory of Computation*. Thomson Course Technology, 2006.

This appendix contains full proofs of the results presented in the text.

A Matching expression and function order with data order (Section 2.3)

In this first section, we consider Lemma 1, which gives a way to translate a program which merely has data order K to one where all sub-expressions in all clauses have a type of order at most K , and where for defined symbols $f : \sigma_1 \Rightarrow \dots \Rightarrow \sigma_n \Rightarrow \kappa$ both each σ_i and κ also have type order $\leq K$.

The work in this appendix may initially seem to be rather more detailed than necessary. However, we must be very precise because we will reuse the proofs to obtain the same results for *arrow depth* and *unitary variables* in Appendix F. To easily combine these proofs, we define:

Definition 20. *In Appendix A, a type σ is proper if $\text{ord}(\sigma) \leq K$ for some fixed non-negative integer K . A program is proper if it is well-formed, and all clauses are typed so that variables are assigned a proper type.*

Note: types of order 0 are proper, and $\sigma \times \tau$ is proper if and only if both σ and τ are proper.

Henceforth, we will refer only to “proper” programs, not to type orders. This allows the lemmas to easily translate to different notions of “proper” (which satisfy the two requirements mentioned as notes.)

A crucial insight to understand the lemma is that if an expression of a certain type is used, then there has to be a variable of at least a matching type order.

Lemma A1. *Given a proper program \mathbf{p} and a derivation tree with root $\mathbf{p}, \gamma \vdash t s_1 \dots s_n \rightarrow w$ (for t an expression), the type of t is $\sigma_1 \Rightarrow \dots \Rightarrow \sigma_n \Rightarrow \tau$ with σ_i proper for all $1 \leq i \leq n$.*

Here, we speak of *the* type of t , because within the context of the derivation, a unique type is associated to all expressions, even variables. Formally, we could for instance consider the type of a variable x to be the type of the value $\gamma(x)$.

Proof. By induction on n ; for $n = 0$ there is nothing to prove. For larger n , note that $t s_1 \dots s_n$ is an application, so the result can only be derived by [Appl]:

$$\frac{\mathbf{p}, \gamma \vdash t s_1 \dots s_{n-1} \rightarrow w_1 \quad \mathbf{p}, \gamma \vdash s_n \rightarrow v_{i+1} \quad \mathbf{p} \vdash^{\text{call}} \mathbf{g} v_1 \dots v_{i+1} \rightarrow w}{\mathbf{p}, \gamma \vdash t s_1 \dots s_n \rightarrow w}$$

Here, $w_1 = \mathbf{g} v_1 \dots v_i$. By the induction hypothesis on the first premise, each of $\sigma_1, \dots, \sigma_{n-1}$ is typed properly; in addition, $v_{i+1} : \sigma_n$ and since $\text{arity}_{\mathbf{p}}(\mathbf{g}) > i$ there must be a rule $\mathbf{g} \ell_1 \dots \ell_{i+1} \dots \ell_k = s$. If σ_n is proper we are done; otherwise, the pattern ℓ_{i+1} can only be a variable or a pair. Since at least one of the immediate subtypes of an improper product type is also improper, ℓ_{i+1} must contain a variable with an improper type; as this contradicts the properness of \mathbf{p} , indeed σ_n is proper. \square

With this insight, we turn to a series of transformations, as used in the proof sketch in the text. The first step—increasing arities of clauses with certain output types—is the hardest. Essentially, we can increase the arities of clauses whose type $\sigma \Rightarrow \tau$ has order $> K$ because, when a value w of type $\sigma \Rightarrow \tau$ is generated, w is eventually applied on some value v of type σ —and not copied before that.

Lemma A2. *Given a proper program \mathbf{p} , let \mathbf{p}' be obtained from \mathbf{p} by replacing all clauses $\mathbf{f} \ell = s$ where s has an improper type $\sigma \Rightarrow \tau$ with σ itself proper, by $\mathbf{f} \ell x = s x$ for some fresh variable x . Then \mathbf{p}' is a well-formed program with data order K , and $\llbracket \mathbf{p} \rrbracket(d_1, \dots, d_M) \mapsto b$ if and only if $\llbracket \mathbf{p}' \rrbracket(d_1, \dots, d_M) \mapsto b$.*

Proof. The preservation of properness is clear, since x can only have a proper type σ and well-formedness is preserved because all clauses with root symbol \mathbf{f} have the same type, so are affected in the same way.

First, if $\llbracket \mathbf{p}' \rrbracket(d_1, \dots, d_M) \mapsto b$ then $\llbracket \mathbf{p} \rrbracket(d_1, \dots, d_M) \mapsto b$ follows easily by induction on the size of the derivation tree of $\llbracket \mathbf{p}' \rrbracket(d_1, \dots, d_M) \mapsto b$; the only non-trivial step, an [Appl] where the third premise uses one of the altered clauses, is handled by using the original rule instead and shifting the subtrees around.

If $\llbracket \mathbf{p} \rrbracket(d_1, \dots, d_M) \mapsto b$, we get $\llbracket \mathbf{p}' \rrbracket(d_1, \dots, d_M) \mapsto b$ by proving by induction on the derivation that $\mathbf{p}', \gamma \vdash s \ t_1 \cdots t_n \rightarrow w_n$ if the following properties hold:

- $s : \sigma_1 \Rightarrow \dots \Rightarrow \sigma_n \Rightarrow \tau$ with τ a proper type;
- $t_1 : \sigma_1, \dots, t_n : \sigma_n$ are expressions and $v_1 : \sigma_1, \dots, v_n : \sigma_n$ are values;
- w_0, \dots, w_n are values with each $w_i : \sigma_{i+1} \Rightarrow \dots \Rightarrow \sigma_n \Rightarrow \tau$;
- $\mathbf{p}, \gamma \vdash s \rightarrow w_0$;
- both $\mathbf{p}', \gamma \vdash t_i \rightarrow v_i$ and $\mathbf{p}' \vdash^{\text{call}} w_{i-1} \ v_i \rightarrow w_i$ for all $1 \leq i \leq n$.

This gives the required result for $n = 0$ and $s = \mathbf{f}_1 \ x_1 \cdots x_M$ (which has a type of order 0). Consider the rule used to derive $\mathbf{p}, \gamma \vdash s \rightarrow w_0$.

Instance $w_0 = \gamma(s)$; then also $\mathbf{p}', \gamma \vdash s \rightarrow w_0$, and we complete with [Appl].

Constructor $n = 0$ and we complete with the induction hypothesis and [Constructor] (all sub-expressions s_i have a type of order 0, which is proper).

Pair $n = 0$; we complete with the induction hypothesis and [Pair] (as the direct subtypes of a proper product types are also proper).

Choice $s = \text{choose } s_1 \cdots s_m$ and the immediate subtree has root $\mathbf{p}, \gamma \vdash s_j \rightarrow w_0$ for some i ; by the induction hypothesis, $\mathbf{p}', \gamma \vdash s_j \ t_1 \cdots t_n \rightarrow w_n$. If $n = 0$, then $\mathbf{p}', \gamma \vdash s \rightarrow w_n$ by [Choice]. Otherwise, $\mathbf{p}', \gamma \vdash s_j \ t_1 \cdots t_n \rightarrow w_n$ can only be obtained by [Appl]; thus, there are $w'_0, \dots, w'_n = w_n$ and v'_1, \dots, v'_n such that $\mathbf{p}', \gamma \vdash s_j \rightarrow w'_0$ and $\mathbf{p}', \gamma \vdash t_i \rightarrow v'_i$ and $\mathbf{p}', \gamma \vdash w'_{i-1} \ v'_i \rightarrow w'_i$ for $1 \leq i \leq n$. This gives first $\mathbf{p}', \gamma \vdash s \rightarrow w'_0$ by [Choice] and then $\mathbf{p}', \gamma \vdash s \ t_1 \cdots t_n \rightarrow w'_n = w_n$ by n uses of [Appl].

Conditional Whether obtained by [If-True] or [If-False], this follows like [Choice].

Function $s = \mathbf{f}$ and the immediate subtree has root $\mathbf{p} \vdash^{\text{call}} \mathbf{f} \rightarrow w_0$.

- If $\text{arity}_{\mathbf{p}}(\mathbf{f}) > 0$, then $w_0 = \mathbf{f}$ and since $\text{arity}_{\mathbf{p}'}(\mathbf{f}) \geq \text{arity}_{\mathbf{p}}(\mathbf{f})$, also $\mathbf{p}', \gamma \vdash s \rightarrow w_0$; we complete with [Appl].
- If $\text{arity}_{\mathbf{p}}(\mathbf{f}) = \text{arity}_{\mathbf{p}'}(\mathbf{f}) = 0$, then $\mathbf{p} \vdash^{\text{call}} \mathbf{f} \rightarrow w_0$ holds because $\mathbf{p}, [] \vdash t \rightarrow w_0$ for some clause $\mathbf{f} = t$. Observing that $\mathbf{p}, [x_1 := v_1, \dots, x_n := v_n] \vdash$

$t \ x_1 \cdots x_n \rightarrow w_n$ by the induction hypothesis, we follow the reasoning from the [Choice] case to obtain $w'_0, \dots, w'_n = w_n$ such that $\mathbf{p}', \gamma \vdash s \rightarrow w'_0$ and $w'_{i-1} \ v_i \rightarrow w'_i$ for $1 \leq i \leq n$ (here, the v_i are unaltered since variables can only be evaluated in one way); we complete with [Appl] once more.

- If $\text{arity}_{\mathbf{p}}(\mathbf{f}) = 0$ but $\text{arity}_{\mathbf{p}'}(\mathbf{f}) = 1$, then $n > 0$ since only clauses with an improper type were altered. As $\mathbf{p}, [] \vdash t \rightarrow w_0$ for some clause $\mathbf{f} = t$, the induction hypothesis gives $\mathbf{p}, [x_1 := v_1, \dots, x_n := v_n] \vdash t \ x_1 \cdots x_n \rightarrow w_n$. As in the [Choice] case, but considering $t \ x_1$ as the head, we find $w'_1, \dots, w'_n = w_n$ such that $\mathbf{p}', [x_1 := v_1, \dots, x_n := v_n] \vdash t \ x_1 \rightarrow w'_1$ and $w'_{i-1} \ v_i \rightarrow w'_i$ for $1 < i \leq n$. Since x_2, \dots, x_n do not occur in $t \ x_1$, we can adapt the first of these trees to have a root $\mathbf{p}', [x_1 := v_1] \vdash t \ x_1 \rightarrow w'_1$. Then we obtain $\mathbf{p}', \gamma \vdash \mathbf{f} \ t_1 \rightarrow w'_1$ from the three subtrees $\mathbf{p}', \gamma \vdash \mathbf{f} \rightarrow \mathbf{f}$ (obtained using [Function] and [Closure]), $\mathbf{p}', \gamma \vdash t_1 \rightarrow v_1$ and $\mathbf{p}' \vdash^{\text{call}} \mathbf{f} \ v_1 \rightarrow w'_1$ (obtained using [Call] from $\mathbf{p}', [x_1 := v_1] \vdash t \ x_1 \rightarrow w'_1$). Using this, $s \ v_1 \ v_2 \cdots v_n \rightarrow w_n$ follows by [Appl].

Appl $s = s_1 \ s_2$ and the immediate subtrees have roots $\mathbf{p}, \gamma \vdash s_1 \rightarrow \mathbf{g} \ v'_1 \cdots v'_j$ and $\mathbf{p}, \gamma \vdash s_2 \rightarrow v_0$ and $\mathbf{p} \vdash^{\text{call}} \mathbf{g} \ v'_1 \cdots v'_j \ v_0 \rightarrow w_0$. By Lemma A1, s_2 has a proper type, so we obtain $\mathbf{p}', \gamma \vdash s_2 \rightarrow v_0$ by the induction hypothesis. Now, using the same reasoning as with [Function], we obtain $w'_0, \dots, w'_n = w_n$ such that $\mathbf{p}' \vdash^{\text{call}} \mathbf{g} \ v'_1 \cdots v'_j \ v_0 \rightarrow w'_0$ and $\mathbf{p}' \vdash^{\text{call}} w'_{i-1} \ v_i \rightarrow w'_i$ for $1 \leq i \leq n$ (if $\text{arity}_{\mathbf{p}}(\mathbf{g}) = j + 1$ and $\text{arity}_{\mathbf{p}'}(\mathbf{g}) = j + 2$, then $w'_0 = \mathbf{g} \ v'_1 \cdots v'_j \ v_0$). Now we may use the induction hypothesis on the subtree $\mathbf{p}, \gamma \vdash s_1 \rightarrow \mathbf{g} \ v'_1 \cdots v'_j$ to obtain $\mathbf{p}', \gamma \vdash s_1 \ s_2 \ t_1 \cdots t_n \rightarrow w_n$. \square

Repeating the transformation of Lemma A2 until it is no longer applicable, we obtain a proper program where all clauses either have a proper type, or a type $\sigma_{k+1} \Rightarrow \dots \Rightarrow \sigma_m \Rightarrow \kappa$ where σ_{k+1} is improper; these latter clauses will be removed in one of the following steps.

The next step is the removal of **if** and **choose** expressions at the head of an application. This is straightforward, and does not depend on properness.

Lemma A3. *Let \mathbf{p}' be obtained from \mathbf{p} by replacing all occurrences of expressions $(\text{if } s_1 \text{ then } s_2 \text{ else } s_3) \ t_1 \cdots t_n$ with $n > 0$ in the right-hand side of any clause by $\text{if } s_1 \text{ then } (s_2 \ t_1 \cdots t_n) \text{ else } (s_3 \ t_1 \cdots t_n)$, and by similarly replacing occurrences of $(\text{choose } s_1 \cdots s_m) \ t_1 \cdots t_n$ with $n > 0$ by $\text{choose } (s_1 \ t_1 \cdots t_n) \cdots (s_m \ t_1 \cdots t_n)$.*

Then, if \mathbf{p} is a proper program also \mathbf{p}' is a proper program and $\llbracket \mathbf{p} \rrbracket(d_1, \dots, d_M) \mapsto b$ iff $\llbracket \mathbf{p}' \rrbracket(d_1, \dots, d_M) \mapsto b$.

Proof. Let $\text{fix}(s)$ be the result of replacing all sub-expressions of the form $(\text{if } b \text{ then } s_1 \text{ else } s_2) \ t_1 \cdots t_n$ in s by $\text{if } b \text{ then } (s_1 \ t_1 \cdots t_n) \text{ else } (s_2 \ t_1 \cdots t_n)$, and expressions $(\text{choose } s_1 \cdots s_m) \ t_1 \cdots t_n$ by $\text{choose } (s_1 \ t_1 \cdots t_n) \cdots (s_m \ t_1 \cdots t_n)$. Then we see, by induction on the size of the derivation tree, that:

- $\mathbf{p} \vdash^{\text{call}} \mathbf{f} \ v_1 \cdots v_n \rightarrow w$ iff $\mathbf{p}' \vdash^{\text{call}} \mathbf{f} \ v_1 \cdots v_n \rightarrow w$, and
- $\mathbf{p}, \gamma \vdash s \rightarrow w$ iff $\mathbf{p}, \gamma \vdash \text{fix}(s) \rightarrow w$.

The case where s has one of the fixable forms merely requires swapping some subtrees. Typing is clearly not affected, nor the other properties of well-formedness, and properness is unaltered because variables are left alone. \square

The effect of this step is to remove sub-expressions of a functional type which, essentially, occur at the head of an application (and therefore have a larger type than is necessary). Since an expression `if b then s_1 else s_2` or `$t = \text{choose } s_1 \cdots s_m$` shares the type of each s_i , this transformation guarantees that the *outermost* expression of a given improper type σ in a clause cannot occur as the direct sub-expression of an `if then else` or `choose`. Thus, in a clause `$f \ell_1 \cdots \ell_k = s$` such an outermost expression is either s itself, or is s_i in some context `$a s_1 \cdots s_n$` with $a \in \mathcal{D} \cup \mathcal{V}$. Following the transformation of Lemma A2, the former situation can only occur if f has a type $\sigma_1 \Rightarrow \dots \Rightarrow \sigma_m \Rightarrow \kappa$ where some σ_i or κ is improper—symbols which we will remove in the final transformation.

Before that, however, we perform one further modification: we alter clauses to remove those sub-expressions which cannot be used following Lemma A1: if `$t s_1 \cdots s_n$` occurs in the right-hand side of a clause and some s_i has an improper type, then this sub-expression can never occur in a derivation tree. Either the clause itself is never used, or the sub-expression occurs below an `if` or `choose` which is never selected. Thus, we can safely replace those sub-expressions by a fresh, unusable symbol. This is done in Lemma A4.

Lemma A4. *Given a proper program p , such that for all clauses `$f \ell = s$` there is no sub-expression `(if s_1 then s_2 else s_3) $t_1 \cdots t_n$` or `(choose $s_1 \cdots s_m$) $t_1 \cdots t_n$` with $n > 0$, let p' be obtained from p by altering all clauses `$f \ell_1 \cdots \ell_k = s$` as follows: if $s \sqsupseteq a s_1 \cdots s_n =: t$ where $a \in \mathcal{V} \cup \mathcal{D}$ and some s_i has an improper type although t itself has a proper type, and t is the leftmost outermost such sub-expression, then replace t in the clause by a fresh symbol \perp_σ , typed $\perp_\sigma : \sigma$, and add a clause $\perp_\sigma = \perp_\sigma$ (to ensure $\perp_\sigma \in \mathcal{D}$ with arity 0).*

Then p' is proper, and $\llbracket p \rrbracket(d_1, \dots, d_M) \mapsto b$ iff $\llbracket p' \rrbracket(d_1, \dots, d_M) \mapsto b$.

Proof. Replacing a sub-expression by a different one of the same type (but potentially fewer variables) cannot affect well-formedness, and as variables are left alone, properness of the variables types is not altered. For the “only if” part, note that the derivation tree for $\llbracket p \rrbracket(d_1, \dots, d_M) \mapsto b$ has no subtree with root $p, \gamma \vdash a s_1 \cdots s_n \rightarrow w$ by Lemma A1. Therefore, `$a s_1 \cdots s_n$` occurs only as a *strict sub-expression* of expressions in the tree for $\llbracket p \rrbracket(d_1, \dots, d_M) \mapsto b$, and may be replaced in all these places by \perp_σ without consequence to obtain a derivation for $\llbracket p' \rrbracket(d_1, \dots, d_M) \mapsto b$. Similarly, for the “if” part, the derivation tree for $\llbracket p' \rrbracket(d_1, \dots, d_M) \mapsto b$ cannot have a subtree with root $p', \gamma \vdash \perp_\sigma \rightarrow w$ since the only clause for \perp_σ does not allow for such a conclusion. Nor can the new clause $\perp_\sigma = \perp_\sigma$ be used in it at all due to non-termination. \square

Note that the transformation from Lemma A4 is terminating, as the size of the affected clause decreases; thus, it can be repeated until no sub-expressions of the given form remain. This gives step 3 of the proof sketch of Lemma 1.

All in all, after these first three steps we still have a well-formed program of the same data order, such that $\llbracket p \rrbracket(d_1, \dots, d_M) \mapsto b$ can be derived for exactly the same d_1, \dots, d_M, b . For step 4, we observe that the offending symbols do not occur in any other clauses anymore.

Lemma A5. *Assume given a proper program \mathbf{p} such that for all clauses $\mathbf{f} \ell_1 \cdots \ell_k = s$ in \mathbf{p} :*

1. *if $\mathbf{f} : \sigma_1 \Rightarrow \dots \Rightarrow \sigma_m \Rightarrow \kappa \in \mathcal{F}$ and all σ_i and κ are proper, then $\sigma_{k+1} \Rightarrow \dots \Rightarrow \sigma_m \Rightarrow \kappa$ is proper as well;*
2. *s does not have a sub-expression of the form $(\mathbf{if} \ s_1 \ \mathbf{then} \ s_2 \ \mathbf{else} \ s_3) \ t_1 \cdots t_n$ or $(\mathbf{choose} \ s_1 \cdots s_m) \ t_1 \cdots t_n$ with $n > 0$;*
3. *s does not have a sub-expression $t = a \ s_1 \cdots s_n$ with $a \in \mathcal{V} \cup \mathcal{D}$ where t itself has a proper type, but with some s_i having an improper type.*

Let Bad be the set of defined symbols \mathbf{g} which are assigned a type $\sigma_1 \Rightarrow \dots \Rightarrow \sigma_m \Rightarrow \kappa$ such that some σ_i or κ is improper. Then for all clauses $\mathbf{f} \ell_1 \cdots \ell_k = s$ in \mathbf{p} with $\mathbf{f} \notin \text{Bad}$: none of the symbols in Bad occur in s , and s does not have any sub-expressions whose type has an order $> K$.

Note that assumption 1 is given by the transformation of Lemma A2, assumption 2 is given by the transformation of Lemma A3, and assumption 3 is given by the transformation of Lemma A4.

Proof. We first observe: if $\mathbf{f} \notin \text{Bad}$, then by the first assumption, the type of s is proper. For such s , which moreover does not have **if** or **choose** expressions at the head of an application, we prove by induction that s does not use elements of Bad . If $s = \mathbf{if} \ s_1 \ \mathbf{then} \ s_2 \ \mathbf{else} \ s_3$ or $s = \mathbf{choose} \ s_1 \cdots s_m$, then each s_i has a proper type (either the type of s or **bool**), so we complete by induction. If $s = \mathbf{c} \ s_1 \cdots s_m$ with $\mathbf{c} \in \mathcal{C}$, each s_i has a type of order 0, which is proper. Otherwise, $s = a \ s_1 \cdots s_n$ with $a \in \mathcal{V} \cup \mathcal{D}$. By the third assumption, all s_i have a proper type, so no bad symbols occur in them by the induction hypothesis. Moreover, if $a \in \mathcal{D}$ and $a : \tau_1 \Rightarrow \dots \Rightarrow \tau_n \Rightarrow \pi \in \mathcal{F}$, then each τ_i is proper by that same assumption, and π is the type of s , so is proper. Thus, also $a \notin \text{Bad}$. \square

Now, a trivial induction shows that the derivation of any conclusion of the form $\llbracket \mathbf{p} \rrbracket(d_1, \dots, d_M) \mapsto b$ cannot use a clause with a bad root symbol; removing these clauses therefore has no effect. As the bad symbols do not occur at all in the remaining clauses, the symbols can also be safely removed. We conclude:

Lemma 1. *Given a well-formed program \mathbf{p} with data order K , there is a well-formed program \mathbf{p}' such that $\llbracket \mathbf{p} \rrbracket(d_1, \dots, d_M) \mapsto b$ iff $\llbracket \mathbf{p}' \rrbracket(d_1, \dots, d_M) \mapsto b$ for any b_1, \dots, b_M, d and: (a) all defined symbols in \mathbf{p}' have a type $\sigma_1 \Rightarrow \dots \Rightarrow \sigma_m \Rightarrow \kappa$ such that both $\text{ord}(\sigma_i) \leq K$ for all i and $\text{ord}(\kappa) \leq K$, and (b) in all clauses, all sub-expressions of the right-hand side have a type of order $\leq K$ as well.*

Proof. We apply the transformations from Lemmas A2–A4 and then remove all “bad” symbols and corresponding clauses following Lemma A5, as described in the text above. As we have seen, the resulting program \mathbf{p}' is still proper, which means that it is well-formed and has the same data order K ; in addition, it has properties (a) and (b) by Lemma A5. \square

In Appendix F we will use variations of Lemma 1 for other notions of “proper”.

B Properties of cons-free programs (Section 3)

Lemma 2 demonstrates that any data encountered during the execution of a cons-free program was either part of the input, or occurred directly as part of a clause; that is, every such data expression is in the set $\mathcal{B}_{d_1, \dots, d_M}^p$.

As a helper result, we start by proving that *patterns* occurring in clauses can only be instantiated to data. This is important to demonstrate the harmlessness of allowing sub-expressions of the left-hand sides of clauses to occur on the right.

Lemma B6. *Let T be a derivation tree with root $p, \gamma \vdash s \rightarrow w$. If s a pattern, then $s\gamma = w$.*

Proof. By induction on the form of T . The roots of [Function], [Choice] and [Conditional] have the wrong shape. [Instance] immediately gives the required result, and the cases for [Constructor] and [Pair] follow by the induction hypothesis.

Finally, we show by induction on n that [Appl] is not applicable: if $n = 1$, then [Appl] requires a subtree $p, \gamma \vdash c \rightarrow f$ with $f \in \mathcal{D}$, for which there are no inference rules. If $n > 1$, then [Appl] requires a subtree $p, \gamma \vdash c s_1 \dots s_{n-1} \rightarrow f v_1 \dots v_i$ which, by the induction hypothesis, must be obtained by an inference rule other than [Appl]; again, there are no suitable inference rules. \square

Rather than immediately proving Lemma 2, we will present—in Lemma B7—a variation which gives a little more information. As this result will be used in some of the later proofs in the appendix, it pays to be precise. First we define a notion of value which is limited to data in $\mathcal{B}_{d_1, \dots, d_M}^p$.

Definition 21. *Fixing a program p and data expressions d_1, \dots, d_M , let the set $\text{Value}_{d_1, \dots, d_M}^p$ be given by the grammar:*

$$v, w \in \text{Value}_{d_1, \dots, d_M}^p ::= d \in \mathcal{B}_{d_1, \dots, d_M}^p \mid (v, w) \mid f v_1 \dots v_n \ (n < \text{arity}_p(f))$$

Note that clearly $\text{Value} \subseteq \text{Value}_{d_1, \dots, d_M}^p$. The following lemma makes the notion “the only data expressions encountered during the execution of a cons-free program p are in $\mathcal{B}_{d_1, \dots, d_M}^p$ ” precise, by requiring all values to be in $\text{Value}_{d_1, \dots, d_M}^p$:

Lemma B7. *Let T be a derivation tree for $\llbracket p \rrbracket(d_1, \dots, d_M) \mapsto b$. Then for all subtrees T' of T :*

- if T' has root $p, \gamma \vdash s \rightarrow w$, then both w and all $\gamma(x)$ are in $\text{Value}_{d_1, \dots, d_M}^p$;
- if T' has root $p, \gamma \vdash^{\text{if}} d, s_1, s_2 \rightarrow w$, then $d \in \mathcal{B}_{d_1, \dots, d_M}^p$ and both w and all $\gamma(x)$ are in $\text{Value}_{d_1, \dots, d_M}^p$;
- if T' has a root $p \vdash^{\text{call}} f v_1 \dots v_n \rightarrow w$ with $f \in \mathcal{D}$, then both w and all v_i are in $\text{Value}_{d_1, \dots, d_M}^p$;
- if T' has a root $p, \gamma \vdash c s_1 \dots s_m \rightarrow c b_1 \dots b_m$ with $c \in \mathcal{C}$, then each $s_i\gamma = b_i \in \text{Data}$ and $c b_1 \dots b_m \in \mathcal{B}_{d_1, \dots, d_M}^p$.

Proof. For brevity, let $\mathcal{B} := \mathcal{B}_{d_1, \dots, d_M}^p$. We show by induction on the depth of T' that: (**) the properties in the lemma statement hold for both T' and all its strict subtrees if $\text{root}(T')$ has one of the following forms:

- $\mathbf{p}, \gamma \vdash s \rightarrow w$ with all $\gamma(x) \in \text{Value}_{d_1, \dots, d_M}^{\mathbf{p}}$, and $t\gamma \in \mathcal{B}$ for all sub-expressions $t \trianglelefteq s$ such that $t = \mathbf{c} \ s_1 \cdots s_m$ for some $\mathbf{c} \in \mathcal{C}$;
- $\mathbf{p}, \gamma \vdash^{\text{if}} d, s_1, s_2 \rightarrow w$ with $d \in \mathcal{B}$ and all $\gamma(x) \in \text{Value}_{d_1, \dots, d_M}^{\mathbf{p}}$, and $t\gamma \in \mathcal{B}$ for all $t \trianglelefteq s_1$ or $t \trianglelefteq s_2$ such that $t = \mathbf{c} \ s_1 \cdots s_m$ for some $\mathbf{c} \in \mathcal{C}$;
- $\mathbf{p} \vdash^{\text{call}} f \ v_1 \cdots v_n \rightarrow w$ with all $v_i \in \text{Value}_{d_1, \dots, d_M}^{\mathbf{p}}$.

Note that proving this suffices: the immediate subtree T' of T has a root $\mathbf{p}, \gamma \vdash \mathbf{f}_1 \ x_1 \cdots x_M \rightarrow b$, where each $\gamma(x_i) = d_i \in \mathcal{B}$, and $\mathbf{f}_1 \ x_1 \cdots x_M$ has no sub-expressions with a data constructor at the head. Thus, the lemma holds for both T' and all its strict subtrees, which implies that it holds for T .

We prove (**). Assume that $\text{root}(T')$ has one of the given forms, and consider the rule used to obtain this root.

- Instance** Then T' has a root $\mathbf{p}, \gamma \vdash x \rightarrow \gamma(x)$; the requirement that all $\gamma(y) \in \text{Value}_{d_1, \dots, d_M}^{\mathbf{p}}$ is satisfied by the assumption, and this also gives that the right-hand side $\gamma(x) \in \text{Value}_{d_1, \dots, d_M}^{\mathbf{p}}$.
- Function** Then T' has a root $\mathbf{p}, \gamma \vdash \mathbf{f} \rightarrow v$ and a subtree $\mathbf{p} \vdash^{\text{call}} \mathbf{f} \rightarrow v$; by the induction hypothesis, the properties hold for this subtree, which also implies that $v \in \text{Value}_{d_1, \dots, d_M}^{\mathbf{p}}$ and therefore the properties hold for T' as well.
- Constructor** Then T' has a root $\mathbf{p}, \gamma \vdash \mathbf{c} \ s_1 \cdots s_m \rightarrow \mathbf{c} \ b_1 \cdots b_m$ with $\mathbf{c} \in \mathcal{C}$, and the immediate subtrees have the form $\mathbf{p}, \gamma \vdash s_i \rightarrow b_i$; by the induction hypothesis (and the assumption), the properties are satisfied for each such subtree. Also by the assumption, $(\mathbf{c} \ s_1 \cdots s_m)\gamma \in \mathcal{B}$, so necessarily each $s_i\gamma \in \mathcal{B} \subseteq \text{Data}$. By Lemma B6, each $s_i\gamma = b_i$, and $\mathbf{c} \ b_1 \cdots b_m = (\mathbf{c} \ s_1 \cdots s_m)\gamma \in \mathcal{B}$.
- Pair** Then T' has a root $\mathbf{p}, \gamma \vdash (s_1, s_2) \rightarrow (w_1, w_2)$ and subtrees with roots $\mathbf{p}, \gamma \vdash s_1 \rightarrow w_1$ and $\mathbf{p}, \gamma \vdash s_2 \rightarrow w_2$. The assumption and induction hypothesis give that the properties are satisfied for both subtrees, and therefore both w_1 and w_2 are in $\text{Value}_{d_1, \dots, d_M}^{\mathbf{p}}$, giving also $(w_1, w_2) \in \text{Value}_{d_1, \dots, d_M}^{\mathbf{p}}$.
- Choice** Then T' has a root $\mathbf{p}, \gamma \vdash \text{choose } s_1 \cdots s_n \rightarrow v$ and a subtree $\mathbf{p}, \gamma \vdash s_i \rightarrow v$ for some i . By the induction hypothesis, the properties hold for the subtree, and therefore $v \in \text{Value}_{d_1, \dots, d_M}^{\mathbf{p}}$.
- Conditional** Then T' has a root $\mathbf{p}, \gamma \vdash \text{if } s_1 \text{ then } s_2 \text{ else } s_3 \rightarrow w$ and subtrees with roots $\mathbf{p}, \gamma \vdash s_1 \rightarrow d$ and $\mathbf{p}, \gamma \vdash^{\text{if}} d, s_2, s_3 \rightarrow w$. The requirement that all $\gamma(x) \in \text{Value}_{d_1, \dots, d_M}^{\mathbf{p}}$ is satisfied by the assumption, and by both the assumption and the induction hypothesis, the lemma is satisfied for the first subtrees. Thus, $d \in \text{Value}_{d_1, \dots, d_M}^{\mathbf{p}}$; for typing reasons $d \in \mathcal{B}$. We may apply the induction hypothesis on the second subtree, which gives that the lemma is satisfied for it, and that $w \in \text{Value}_{d_1, \dots, d_M}^{\mathbf{p}}$.
- If-True or If-False** Then $\text{root}(T')$ has the form $\mathbf{p}, \gamma \vdash^{\text{if}} d, s_1, s_2 \rightarrow w$. The requirement that $d \in \mathcal{B}$ and all $\gamma(x) \in \text{Value}_{d_1, \dots, d_M}^{\mathbf{p}}$ is satisfied by the assumption. T' has one immediate subtree T'' , whose root is either $\mathbf{p}, \gamma \vdash s_2 \rightarrow w$ or $\mathbf{p}, \gamma \vdash s_3 \rightarrow w$. Since the assumptions are satisfied, T'' satisfies the lemma by the induction hypothesis, which also gives that $w \in \text{Value}_{d_1, \dots, d_M}^{\mathbf{p}}$.
- Appl** Then T' has a root $\mathbf{p}, \gamma \vdash s \ t \rightarrow w$ and subtrees $\mathbf{p}, \gamma \vdash s \rightarrow \mathbf{f} \ v_1 \cdots v_n$ and $\mathbf{p}, \gamma \vdash t \rightarrow v_{n+1}$ and $\mathbf{p} \vdash^{\text{call}} \mathbf{f} \ v_1 \cdots v_{n+1} \rightarrow w$. The assumption gives that all $\gamma(x) \in \text{Value}_{d_1, \dots, d_M}^{\mathbf{p}}$, and the assumption and induction hypothesis together

give that the lemma is satisfied for the first two subtrees. Since this implies that all $v_i \in \text{Value}_{d_1, \dots, d_M}^p$, we may also apply the induction hypothesis on the last subtree, which gives that $w \in \text{Value}_{d_1, \dots, d_M}^p$.

Closure Then $\text{root}(T')$ has the form $\mathbf{p}, \vdash^{\text{call}} \mathbf{f} \ v_1 \cdots v_n \rightarrow w$ with $\mathbf{f} \in \mathcal{D}$. All v_i are in $\text{Value}_{d_1, \dots, d_M}^p$ by the assumption; thus, $w = f \ v_1 \cdots v_n \in \text{Value}_{d_1, \dots, d_M}^p$ as well, and there are no strict subtrees.

Call Then $\text{root}(T')$ has the form $\mathbf{p}, \vdash^{\text{call}} \mathbf{f} \ v_1 \cdots v_k \rightarrow w$ with $\mathbf{f} \in \mathcal{D}$, and there exist a clause $\mathbf{f} \ \ell_1 \cdots \ell_k = s$ and an environment γ with domain $\text{Var}(\mathbf{f} \ \ell_1 \cdots \ell_k)$ such that each $v_i = \ell_i \gamma$, and T' has one immediate subtree T'' with root $\mathbf{p}, \gamma \vdash s \rightarrow w$. Then, for $1 \leq i \leq n$ we observe that $v_i \sqsupseteq \gamma(x)$ for all $x \in \text{Var}(\ell_i)$, since (by definition of a pattern) $\ell_i \sqsupseteq x$ for all such x . Since all sub-expressions of a value in $\text{Value}_{d_1, \dots, d_M}^p$ are themselves in $\text{Value}_{d_1, \dots, d_M}^p$, we thus have: each $\gamma(x) \in \text{Value}_{d_1, \dots, d_M}^p$.

Moreover, for $s \sqsupseteq t = c \ s_1 \cdots s_m$ with $c \in \mathcal{C}$, also $\ell_i \sqsupseteq t$ for some i by definition of cons-free. But then also $\ell_i \gamma = v_i \sqsupseteq t \gamma$. Thus, we can apply the induction hypothesis, and obtain that the lemma is satisfied for T'' . This implies that $w \in \text{Value}_{d_1, \dots, d_M}^p$, so the last requirement on the root of T' is satisfied. \square

It remains to prove Lemma 2 from the text—which is just a (slightly weakened) reformulation of Lemma B7.

Lemma 2. *Let \mathbf{p} be a cons-free program, and suppose that $\llbracket \mathbf{p} \rrbracket(d_1, \dots, d_M) \mapsto b$ is obtained by a derivation tree T . Then for all statements $\mathbf{p}, \gamma \vdash s \rightarrow w$ or $\mathbf{p}, \gamma \vdash^{\text{if}} b', s_1, s_2 \rightarrow w$ or $\mathbf{p} \vdash^{\text{call}} \mathbf{f} \ v_1 \cdots v_n \rightarrow w$ in T , and all expressions t such that (a) $w \sqsupseteq t$, (b) $b' \sqsupseteq t$, (c) $\gamma(x) \sqsupseteq t$ for some x or (d) $v_i \sqsupseteq t$ for some i : if t has the form $c \ b_1 \cdots b_m$ with $c \in \mathcal{C}$, then $t \in \mathcal{B}_{d_1, \dots, d_M}^p$.*

Proof. Immediately by Lemma B7, as the only sub-expressions of an element of $\text{Value}_{d_1, \dots, d_M}^p$ with a data constructor as head symbol, are in $\mathcal{B}_{d_1, \dots, d_M}^p$. \square

C Counting modules (Section 5.1)

We discuss the counting modules from Section 5.1 in more detail. To start, we use the ideas of Example 9 to create counting modules surpassing any polynomial.

Lemma 4. *For any $a, b \in \mathbb{N} \setminus \{0\}$, there is a $(\lambda n. a \cdot (n+1)^b)$ -counting module $C_{\langle a, b \rangle}$ with data order 0.*

Proof. Using pairing in a right-associative way—so (x, y, z) should be read as $(x, (y, z))$ —we let:

- $\alpha_{\langle a, b \rangle} := \text{list}^{b+1}$; that is, $\text{list} \times \cdots \times \text{list}$ with $b+1$ occurrences of list
- $\mathcal{A}_{\langle a, b \rangle}^n := \{(d_0, \dots, d_b) \mid \text{all } d_i \text{ are boolean lists, with } |d_0| < a \text{ and } |d_i| \leq n \text{ for } 1 \leq i \leq b; \text{ here, we say } |x_1 :: \dots :: x_k :: []| = k\}$
- $\langle (d_0, \dots, d_b) \rangle_{\langle a, b \rangle}^n := \sum_{i=0}^b |d_i| \cdot (n+1)^{b-i}$
- $\mathcal{D}_{\langle a, b \rangle} = \{\text{seed}_{\langle a, b \rangle}, \text{pred}_{\langle a, b \rangle}, \text{zero}_{\langle a, b \rangle}\}$

- let **alist** be a list of length $a - 1$, e.g., `false::...::false::[]` and let $\mathbf{p}_{\langle a,b \rangle}$ consist of the following clauses:

```

seed $\langle a,b \rangle$  cs = (alist, cs, ..., cs)

pred $\langle a,b \rangle$  cs (x0, ..., xb-1, y::ys) = (x0, ..., xb-1, ys)
pred $\langle a,b \rangle$  cs (x0, ..., xb-2, y::ys, []) = (x0, ..., xb-2, ys, cs)
...
pred $\langle a,b \rangle$  cs (y::ys, [], ..., []) = (ys, cs, ..., cs)
pred $\langle a,b \rangle$  cs ([], [], ..., []) = ([], [], ..., [])

zero $\langle a,b \rangle$  cs (x0, ..., xb-1, y::ys) = false
zero $\langle a,b \rangle$  cs (x0, ..., xb-2, y::ys, []) = false
...
zero $\langle a,b \rangle$  cs (y::ys, [], ..., []) = false
zero $\langle a,b \rangle$  cs ([], ..., []) = true

```

It is easy to see that the requirements on evaluation are satisfied. For example, $\mathbf{p}_{\langle a,b \rangle} \vdash^{\text{call}} \mathbf{seed}_{\langle a,b \rangle} \text{cs} \rightarrow (\mathbf{alist}, \text{cs}, \dots, \text{cs})$, which consists of $b+1$ boolean lists with the right lengths, and $\langle (\mathbf{alist}, \text{cs}, \dots, \text{cs}) \rangle_{\langle a,b \rangle}^n = (a-1) \cdot (n+1)^b + n \cdot (n+1)^{b-1} + \dots + n \cdot (n+1)^{b-b} = (a \cdot (n+1)^b - (n+1)^b) + ((n+1)^b - (n+1)^{b-1}) + \dots + ((n+1)^1 - (n+1)^0) = a \cdot (n+1)^b - 1$; as the program is deterministic, this is the only possible result. The requirements for $\mathbf{pred}_{\langle a,b \rangle}$ and $\mathbf{zero}_{\langle a,b \rangle}$ are similarly easy. \square

Note that the clauses in $\mathbf{p}_{\langle a,b \rangle}$ correspond to those in Example 9. The other counting module of Section 5.1 allows us to build on an existing counting module so as to obtain an exponential increase in magnitude of the boundary P —and to be applied repeatedly for arbitrarily high bounds.

Lemma 5. *If there is a P -counting module C_π of data order K , then there is a $(\lambda n. 2^{P(n)})$ -counting module $C_{\mathbf{e}[\pi]}$ of data order $K+1$.*

Proof. We let:

- $\alpha_{\mathbf{e}[\pi]} := \alpha_\pi \Rightarrow \text{bool}$; then $\text{ord}(\alpha_{\mathbf{e}[\pi]}) \leq K+1$;
- $\mathcal{A}_{\mathbf{e}[\pi]}^n := \{\text{values } F \text{ such that, (a) for all } v \in \mathcal{A}_\pi^n: \text{either } \mathbf{p}_{\mathbf{e}[\pi]} \vdash^{\text{call}} F v \rightarrow \text{true} \text{ or } \mathbf{p}_{\mathbf{e}[\pi]} \vdash^{\text{call}} F v \rightarrow \text{false} \text{ (but not both), and (b) for all } v, w \in \mathcal{A}_\pi^n: \text{if } \langle v \rangle_\pi^n = \langle w \rangle_\pi^n \text{ then } \mathbf{p}_{\mathbf{e}[\pi]} \vdash^{\text{call}} F v \rightarrow b \text{ and } \mathbf{p}_{\mathbf{e}[\pi]} \vdash^{\text{call}} F w \rightarrow d \text{ implies } b = d\}; \text{ that is, } \mathcal{A}_{\mathbf{e}[\pi]}^n \text{ is the set of functions from } \alpha_\pi \text{ to } \text{bool} \text{ such that } F[i] \text{ is uniquely defined for any representation } [i] \text{ of } i \in \{0, \dots, P(n)-1\} \text{ in } C_\pi;$
- $\langle F \rangle_{\mathbf{e}[\pi]}^n = \sum_{i=0}^{P(n)-1} \{2^{P(n)-1-i} \mid \exists v \in \mathcal{A}_\pi^n [\langle v \rangle_\pi^n = i \wedge \mathbf{p}_{\mathbf{e}[\pi]} \vdash^{\text{call}} F i \rightarrow \text{true}]\}$; that is, F is mapped to the number i with a bitstring $b_0 \dots b_{P(n)-1}$, where $b_i = 1$ if and only if $F[i]$ has value **true**;
- $\mathcal{D}_{\mathbf{e}[\pi]} = \mathcal{D}_\pi \cup \{\text{not}\} \cup \{\mathbf{f}_{\mathbf{e}[\pi]} \mid \mathbf{f}_{\mathbf{e}[\pi]} \text{ used in } \mathbf{p}_{\mathbf{e}[\pi]} \text{ below}\}$
- $\mathbf{p}_{\mathbf{e}[\pi]}$ consists of the following clauses, followed by the clauses in \mathbf{p}_π :

```

// 2P(n) - 1 corresponds to the bitvector which is 1 at all bits
seed $\mathbf{e}[\pi]$  cs = alwaystrue $\mathbf{e}[\pi]$ 
alwaystrue $\mathbf{e}[\pi]$  x = true

```

```

// to test whether  $b_0 \dots b_{P(n)-1}$  is 0, check each  $b_i = 0$ 
// start in  $b_{P(n)-1}$  and count down to test all bits.
zeroe[π] cs F = zhe[π] cs F (seedπ cs)
zhe[π] cs F k = if F k then false
                elseif zeroπ cs k then true
                else zhe[π] cs F (predπ cs k)

// the predecessor of  $b_0 \dots b_i 10 \dots 0$  is  $b_0 \dots b_i 01 \dots 1$ , so go down
// through the bits, and flip them until you encounter a 1
prede[π] cs F = phe[π] cs F (seedπ cs)
phe[π] cs F k = if F k then flipe[π] cs F k
                elseif zeroπ cs k then seede[π] cs
                else phe[π] cs (flipe[π] cs F k) (predπ cs k)
flipe[π] cs F k i = if equalπ cs k i then not (F i) else F i
not b = if b then false else true

```

By standard bitvector arithmetic, the evaluation requirements are satisfied. \square

D Algorithm complexity (Sections 5.2 and 6.2)

Now, we turn to proving the complexity of both the core and general algorithms (Algorithm 7 and 13). The key result—which is formulated for Algorithm 7 but immediately extends to Algorithm 13, is the first lemma of Section 5.2:

Lemma 8. *Let \mathbf{p} be a cons-free program of data order K . Let Σ be the set of all types σ with $\text{ord}(\sigma) \leq K$ which occur as part of an argument type, or as an output type of some $\mathbf{f} \in \mathcal{D}$. Suppose that, given input of total size n , $\langle \sigma \rangle_{\mathcal{B}}$ has cardinality at most $F(n)$ for all $\sigma \in \Sigma$, and testing whether $e_1 \sqsupseteq e_2$ for $e_1, e_2 \in \llbracket \sigma \rrbracket_{\mathcal{B}}$ takes at most $F(n)$ steps. Then Algorithm 7 runs in $\text{TIME}(a \cdot F(n)^b)$ for some a, b .*

Proof. We first observe that, for any $e \in \llbracket \sigma \rrbracket_{\mathcal{B}}$ occurring in the algorithm, $\sigma \in \Sigma$. This is due to the preparation step where \mathbf{p} is replaced by \mathbf{p}' .

Write \mathbf{a} for the greatest number of arguments any defined symbol \mathbf{f} or variable x occurring in \mathbf{p}' may take, and write \mathbf{r} for the greatest number of sub-expressions of any right-hand side in \mathbf{p}' (which does not depend on the input!). We start by observing that \mathcal{X} contains at most $\mathbf{a} \cdot |\mathcal{D}| \cdot F(n)^{\mathbf{a}+1}$ statements $\mathbf{f} \ e_1 \dots e_n \rightsquigarrow o$, and at most $|\mathbf{p}'| \cdot \mathbf{r} \cdot F(n)^{\mathbf{a}+1}$ statements $t\eta \rightsquigarrow o$.

We observe that step 1a does not depend on the input, so takes a constant number of steps. Step 1b and 1c both take $|\mathcal{X}|$ steps. The exact time cost of each step depends on implementation concerns, but is certainly limited by some polynomial of $F(n)$, by the assumption on \sqsupseteq . Thus, the preparation step is polynomial in $F(n)$; say its cost is $P_1(F(n))$.

In every step of the iteration, at least one statement is flipped from unconfirmed to confirmed, or the iteration ends. Thus, there are at most $|\mathcal{X}| + 1$ iterations. In each iteration, Step 2a has a cost limited by $\text{Card}(O) \cdot |\mathcal{X}| \cdot \langle \text{cost of checking } u' \sqsupseteq u \rangle \leq F(n)^3 \cdot |\mathcal{X}| \cdot \langle \text{some implementation-dependent constant} \rangle$.

Step 2b has a cost limited by $|\mathbf{p}'| \cdot \langle \text{cost of matching} \rangle \cdot |\mathcal{X}| \cdot \langle \text{some implementation-dependent constant} \rangle$. Both Steps 2c and 2e are limited by $2 \cdot \langle \text{some constant} \rangle \cdot |\mathcal{X}|$ as well (the cost for looking up confirmation status of two given statements), and Step 2d is certainly limited by $r \cdot \langle \text{some constant} \rangle \cdot |\mathcal{X}|$.

For each statement $s\eta \leadsto o$ in Steps 2f and 2g, we must check all suitable tuples $(e_1, \dots, e_{n'})$ —of which there are at most $F(n)^a$ —and test confirmation for each $s_i\eta \leadsto e_i$. In Step 2f, we must additionally do \sqsubseteq tests for all $o' \in \eta(x)(e_1, \dots, e_{n'})$ for all tuples; even if we ignore that $\eta(x)$ is a partial function, this takes at most $F(n)^a \cdot F(n)^a \cdot F(n) \cdot \langle \text{some constant} \rangle$ steps. In Step 2(g)i, a single lookup over $|\mathcal{X}|$ statements must be done; in Step 2(g)ii this is combined with a lookup. Both cases certainly stay below $F(n)^{2 \cdot a + 2} \cdot \langle \text{some constant} \rangle$ steps.

In total, the cost of iterating is thus limited by $(|\mathcal{X}| + 1) \cdot |\mathcal{X}| \cdot \langle \text{some constant} \rangle \cdot \max(F(n)^3 \cdot |\mathcal{X}|, |\mathbf{p}'| \cdot |\mathcal{X}|, 2 \cdot |\mathcal{X}|, r \cdot |\mathcal{X}|, F(n)^{2 \cdot a + 2})$. Since $|\mathcal{X}|$ is a polynomial in $F(n)$, this is certainly bounded by $P_2(F(n))$ for some polynomial P_2 .

Finally, completion requires at most $|\mathcal{X}|$ tests. Overall, all steps together gives a polynomial time complexity in $F(n)$. \square

Thus, complexity of Algorithm 7 relies on the size of each $\langle \sigma \rangle_{\mathcal{B}}$, and complexity of Algorithm 13 on the sizes of $\llbracket \sigma \rrbracket_{\mathcal{B}}$.

To determine these sizes as well as the complexity of testing \sqsubseteq on two given extensional values, we first obtain a simple helper lemma for calculation:

Lemma D8. *If $X, Y \geq 2$, then $\exp_2^K(X) \cdot \exp_2^K(Y) \geq \exp_2^K(X \cdot Y)$ for $K \in \mathbb{N}$.*

Proof. We start by observing that for $X, Y \geq 2$ always $(**) X \cdot Y \geq X + Y$:

- $2 \cdot 2 = 4 = 2 + 2$;
- if $X \cdot Y \geq X + Y$, then $X \cdot (Y + 1) = X \cdot Y + X \geq (X + Y) + X \geq X + (Y + 1)$;
- if $X \cdot Y \geq X + Y$, then $(X + 1) \cdot Y \geq Y + X + 1$ in the same way.

By induction on K we also see: $(***)$ if $X \geq 2$ then $\exp_2^K(X) \geq 2$ for all K .

Now the lemma follows by another induction on K :

- for $K = 0$: $\exp_2^K(X) \cdot \exp_2^K(Y) = X \cdot Y = \exp_2^K(X \cdot Y)$;
- for $K \geq 0$: $\exp_2^{K+1}(X) \cdot \exp_2^{K+1}(Y) = 2^{\exp_2^K(X)} \cdot 2^{\exp_2^K(Y)} = 2^{\exp_2^K(X) + \exp_2^K(Y)} \leq 2^{\exp_2^K(X) \cdot \exp_2^K(Y)}$ by $(**)$ and $(***)$, $\leq 2^{\exp_2^K(X) \cdot \exp_2^K(Y)} = \exp_2^{K+1}(X \cdot Y)$ by the induction hypothesis. \square

For the first part of Lemma 9, we consider the cardinality of each $\langle \sigma \rangle_{\mathcal{B}}$.

Lemma D9. *If $1 \leq \text{Card}(\mathcal{B}) < N$, then for each σ with $\text{ord}(\sigma) \leq K$ such that L sorts occur in σ (including repetitions) we have: $\text{Card}(\langle \sigma \rangle_{\mathcal{B}}) < \exp_2^K(N^L)$.*

Proof. By induction on the form of σ .

For $\sigma \in \mathcal{S}$, $\langle \sigma \rangle_{\mathcal{B}} \subseteq \mathcal{B}$ so $\text{Card}(\langle \sigma \rangle_{\mathcal{B}}) \leq \text{Card}(\mathcal{B}) < N$.

For $\sigma = \sigma_1 \times \sigma_2$ with σ_1 having L_1 sorts and σ_2 having L_2 , we have

$$\begin{aligned} \text{Card}(\langle \sigma_1 \times \sigma_2 \rangle_{\mathcal{B}}) &= \text{Card}(\langle \sigma_1 \rangle_{\mathcal{B}}) \cdot \text{Card}(\langle \sigma_2 \rangle_{\mathcal{B}}) \\ &< \exp_2^K(N^{L_1}) \cdot \exp_2^K(N^{L_2}) \\ &\leq \exp_2^K(N^{L_1} \cdot N^{L_2}) \text{ by Lemma D8} \\ &= \exp_2^K(N^{L_1 + L_2}) = \exp_2^K(L) \end{aligned}$$

For $\sigma = \sigma_1 \Rightarrow \sigma_2$ with σ_1 having L_1 sorts and σ_2 having L_2 , each element of $\langle\sigma\rangle_{\mathcal{B}}$ can be seen as a *total* function from $\langle\sigma_1\rangle_{\mathcal{B}}$ to $\langle\sigma_2\rangle_{\mathcal{B}} \cup \{\perp\}$. Therefore,

$$\begin{aligned} \text{Card}(\langle\sigma_1 \Rightarrow \sigma_2\rangle_{\mathcal{B}}) &= (\text{Card}(\langle\sigma_2\rangle_{\mathcal{B}}) + 1)^{\text{Card}(\langle\sigma_1\rangle_{\mathcal{B}})} \\ &\leq \exp_2^K(N^{L_2})^{\text{Card}(\langle\sigma_1\rangle_{\mathcal{B}})} \\ &< \exp_2^K(N^{L_2})^{\wedge(\exp_2^{K-1}(N^{L_1}))} \\ &= 2^{\wedge(\exp_2^{K-1}(N^{L_2}) \cdot \exp_2^{K-1}(N^{L_1}))} \\ &\leq 2^{\wedge(\exp_2^{K-1}(N^L))} \text{ by Lemma D8} \\ &= \exp_2^K(N^L) \end{aligned} \quad \square$$

The cardinality of each $\llbracket\sigma\rrbracket_{\mathcal{B}}$ (as used in Lemma 14) is obtained similarly.

Lemma D10. *If $1 \leq \text{Card}(\mathcal{B}) < N$, then for each σ with $\text{depth}(\sigma) \leq K$ such that L sorts occur in σ (including repetitions) we have: $\text{Card}(\llbracket\sigma\rrbracket_{\mathcal{B}}) < \exp_2^K(N^L)$.*

Proof. By induction on the form of σ .

For $\sigma \in \mathcal{S}$, $\llbracket\sigma\rrbracket_{\mathcal{B}} \subseteq \mathcal{B}$ so $\text{Card}(\llbracket\sigma\rrbracket_{\mathcal{B}}) \leq \text{Card}(\mathcal{B}) < N$.

For $\sigma = \sigma_1 \times \sigma_2$, we obtain $\text{Card}(\llbracket\sigma\rrbracket_{\mathcal{B}}) \leq \exp_2^K(N^L)$ in exactly the same way as in Lemma D9.

For $\sigma = \sigma_1 \Rightarrow \sigma_2$ with σ_1 having L_1 sorts and σ_2 having L_2 , each element of $\llbracket\sigma\rrbracket_{\mathcal{B}}$ is a subset of $\llbracket\sigma_1\rrbracket_{\mathcal{B}} \times \llbracket\sigma_2\rrbracket_{\mathcal{B}}$; therefore,

$$\begin{aligned} \text{Card}(\langle\sigma_1 \Rightarrow \sigma_2\rangle_{\mathcal{B}}) &= 2^{\wedge(\text{Card}(\langle\sigma_1\rangle_{\mathcal{B}} \times \langle\sigma_2\rangle_{\mathcal{B}}))} \\ &\leq 2^{\wedge(\exp_2^{K-1}(N^{L_1}) \cdot \exp_2^{K-1}(N^{L_2}))} \\ &\leq 2^{\wedge(\exp_2^{K-1}(N^L))} \text{ by Lemma D8} \\ &= \exp_2^K(N^L) \end{aligned} \quad \square$$

Aside from the cardinalities of $\langle\sigma\rangle_{\mathcal{B}}$ and $\llbracket\sigma\rrbracket_{\mathcal{B}}$, Lemmas 9 and 14 also consider the complexity of deciding $e \sqsupseteq u$ for two (deterministic or non-deterministic) extensional values. This complexity we consider for both lemmas together:

Lemma D11. *Let $[\sigma]$ be one of $\langle\sigma\rangle_{\mathcal{B}}$ or $\llbracket\sigma\rrbracket_{\mathcal{B}}$, and suppose that we know that for all subtypes of σ containing L sorts: $\text{Card}([\sigma]) < \exp_2^K(N^L)$ for some fixed K , and $N \geq 2$. Then for any $e, u \in [\sigma]$: testing $e \sqsupseteq u$ requires $< \exp_2^K(N^{(L+1)^3})$ comparisons between elements of \mathcal{B} .*

Proof. We let C_{σ} be the maximum cost of either \sqsupseteq tests or equality tests for elements of $[\sigma]$. We first observe:

1. $(X + Y + 1)^3 = X^3 + Y^3 + 3X^2Y + 3XY^2 + 3X^2 + 3Y^2 + 6XY + 3X + 3Y + 1$;
2. $(X + 1)^3 = X^3 + 3X^2 + 3X + 1$;
3. $(X + Y + 1)^3 - (X + 1)^3 - (Y + 1)^3 = 3X^2Y + 3XY^2 + 6XY - 1$.

Now, $C_{\iota} = 1 < N^8 = \exp_2^K(N^{2^3})$ for $\iota \in \mathcal{S}$. Writing L_1 for the number of sorts in σ_1 and L_2 for the number of sorts in σ_2 , we have:

$$\begin{aligned} C_{\sigma_1 \times \sigma_2} &= C_{\sigma_1} + C_{\sigma_2} \\ &< \exp_2^K(N^{(L_1+1)^3}) + \exp_2^K(N^{(L_2+1)^3}) \text{ by the induction hypothesis} \\ &\leq \exp_2^K(N^{(L_1+1)^3} \cdot N^{(L_2+1)^3}) \text{ because both sides are at least 2} \\ &\leq \exp_2^K(N^{(L_1+1)^3 + (L_2+1)^3}) \text{ by Lemma D8} \\ &\leq \exp_2^K(N^{(L_1+L_2+1)^3}) \text{ by observation 3 above} \\ &= \exp_2^K(N^{(L+1)^3}) \end{aligned}$$

To compare $A_{\sigma_1 \Rightarrow \sigma_2}$ and $B_{\sigma_1 \Rightarrow \tau_1}$, we may for instance do the following:

- for all $(u_1, u_2) \in B$:
 - for all $(e_1, e_2) \in A$, test $e_1 = u_1$ and either $e_2 = e_2$ or $e_2 \sqsupseteq u_2$;
 - conclude failure if we didn't find a match
- in the case of \sqsupseteq , conclude success if we haven't concluded failure yet; in the case of $=$, also do the test in the other direction

This gives, roughly:

$$\begin{aligned}
 C_{\sigma \Rightarrow \tau} &\leq 2 \cdot \text{Card}([\sigma_1 \times \sigma_2]) \cdot \text{Card}([\sigma_1 \times \sigma_2]) \cdot (C_{\sigma_1} + C_{\sigma_2}) \\
 &\leq 2 \cdot \exp_2^K(N^L) \cdot \exp_2^K(N^L) \cdot (C_{\sigma_1} + C_{\sigma_2}) \\
 &< 2 \cdot \exp_2^K(N^L) \cdot \exp_2^K(N^L) \cdot \exp_2^K(N^{(L_1+1)^3+(L_2+1)^3}) \text{ as above} \\
 &\leq 2 \cdot \exp_2^K(N^{2 \cdot L + (L_1+1)^3 + (L_2+1)^3}) \text{ by Lemma D8} \\
 &\leq \exp_2^K(N^{2 \cdot L + (L_1+1)^3 + (L_2+1)^3 + 1}) \text{ because } N \geq 2 \\
 &\leq \exp_2^K(N^{(L_1+L_2+1)^3}) \text{ by observation 3 above} \\
 &\quad \text{because } (X + 6L_1L_2 - 1) - (2L_1 + 2L_2 + 1) \geq 0 \text{ when } L_1, L_2 \geq 1 \quad \square
 \end{aligned}$$

All parts now proven, Lemmas 9 and 14 follow immediately.

Lemma 9. *If $1 \leq \text{Card}(\mathcal{B}) < N$, then for each σ of length L (where the length of a type is the number of sorts occurring in it, including repetitions), with $\text{ord}(\sigma) \leq K$: $\text{Card}(\llbracket \sigma \rrbracket_{\mathcal{B}}) < \exp_2^K(N^L)$. Testing $e \sqsupseteq u$ for $e, u \in \llbracket \sigma \rrbracket_{\mathcal{B}}$ takes at most $\exp_2^K(N^{(L+1)^3})$ comparisons between elements of \mathcal{B} .*

Proof. The first part is Lemma D9; using this, the second part follows by Lemma D11. \square

Lemma 14. *If $1 \leq \text{Card}(\mathcal{B}) < N$, then for each σ of length L , with $\text{depth}(\sigma) \leq K$: $\text{Card}(\llbracket \sigma \rrbracket_{\mathcal{B}}) < \exp_2^K(N^L)$. Testing $e \sqsupseteq u$ for $e, u \in \llbracket \sigma \rrbracket_{\mathcal{B}}$ takes at most $\exp_2^K(N^{(L+1)^3})$ comparisons.*

Proof. The first part is Lemma D10; using this, the second part follows by Lemma D11. \square

E Algorithm correctness (Section 6.3)

We prove that for both Algorithm 7 and Algorithm 13: $\llbracket \mathbf{p} \rrbracket(d_1, \dots, d_M) \mapsto b$ if and only if b is in the set returned by the algorithm. We do this in four steps:

Section E.1 we obtain some properties on (deterministic or non-deterministic) extensional values and \mathbf{p}' ;

Section E.2 we prove that for both algorithms: if b is returned by the algorithm, then $\llbracket \mathbf{p} \rrbracket(d_1, \dots, d_M) \mapsto b$;

Section E.3 we prove that for Algorithm 13: if $\llbracket \mathbf{p} \rrbracket(d_1, \dots, d_M) \mapsto b$, then b is returned by the algorithm;

Section E.4 we adapt this proof to the deterministic setting.

In this, we break from the order in the main text: where the text considers the deterministic case first (Algorithm 7), we will show completeness first for the non-deterministic case (Algorithm 13). The reason for this choice is that our algorithm has been designed particularly for the non-deterministic cases (both the general non-deterministic setting which results in a classification of **ELEMENTARY**, and the result for arrow depth in Section 7) for which no algorithm yet existed in the literature. This results in a significantly simpler proof.

We do also handle the deterministic case, but this requires an extra proof step to replace the sets $\llbracket \sigma \rrbracket_{\mathcal{B}}$ of non-deterministic extensional values by the sets $\langle \sigma \rangle_{\mathcal{B}}$ of deterministic extensional values.

Note that all deterministic extensional values are also non-deterministic extensional values. In this appendix, *extensional values* may refer to either deterministic or non-deterministic extensional values.

E.1 Properties of extensional values and \mathbf{p}'

We begin by deriving some properties relevant to both the soundness and completeness proofs. First, the following lemma will be invaluable when matching extensional values against the left-hand sides of clauses.

Lemma E12. *Fix a set \mathcal{B} of data expressions, closed under taking sub-expressions. Let \Downarrow be a relation, relating values v of type σ to extensional values $e \in \llbracket \sigma \rrbracket_{\mathcal{B}}$, notation $v \Downarrow e$, such that:*

- $v \Downarrow e$ for v, e data if and only if $v = e$, and
- $(v, w) \Downarrow (e, u)$ if and only if both $v \Downarrow e$ and $w \Downarrow u$.

Let $v_1 : \sigma_1, \dots, v_k : \sigma_k$ and $e_1 \in \llbracket \sigma_1 \rrbracket_{\mathcal{B}}, \dots, e_k \in \llbracket \sigma_k \rrbracket_{\mathcal{B}}$ be such that $s_i \Downarrow e_i$ for each i , and let $\rho : \mathbf{f} \ell_1 \cdots \ell_k = s$ be a clause. Then there is an environment γ such that each $v_i = \ell_i \gamma$ if and only if there is an ext-environment η such that each $e_i = \ell_i \eta$, and if both are satisfied then $\gamma(x) \Downarrow \eta(x)$ for all $x \in \text{Var}(\mathbf{f} \ell_1 \cdots \ell_k)$.

Essentially, this lemma says that no matter how we associate values of a higher type to extensional values, if data and pairing are handled as expected, then matching is done in the natural way.

Proof. For ℓ a pattern of type σ , $v : \sigma$ a value and $e \in \llbracket \sigma \rrbracket_{\mathcal{B}}$ such that $v \Downarrow e$, the lemma follows easily once we prove the following by induction on ℓ :

- If $v = \ell \gamma$ for some γ , then there exists η on domain $\text{Var}(\ell)$ such that $e = \ell \eta$ and $\gamma(x) \Downarrow \eta(x)$ for all x in the domain:
 - If ℓ is a variable, then $\gamma(\ell) = v$, so choose $\eta := [\ell := e]$.
 - If ℓ is a pair (ℓ_1, ℓ_2) , then $v = (v_1, v_2)$ and therefore $e = (e_1, e_2)$ with both $v_1 \Downarrow e_1$ and $v_2 \Downarrow e_2$; by the induction hypothesis, we find η_1 and η_2 on domains $\text{Var}(\ell_1)$ and $\text{Var}(\ell_2)$ respectively; we are done with $\eta := \eta_1 \cup \eta_2$.
 - If $\ell = \mathbf{c} \ell_1 \cdots \ell_m$ with $\mathbf{c} \in \mathcal{C}$, then v and e are both data expressions, so $v = e$; since the argument types of constructors have order 0, all $x \in \text{Var}(\ell)$ have type order 0, so we can choose $\eta(x) := \gamma(x)$ for such x .
- If $e = \ell \eta$ for some η , then there exists γ on domain $\text{Var}(\ell)$ such that $s = \ell \gamma$ and $\gamma(x) \Downarrow \eta(x)$ for x in $\text{Var}(\ell)$; this reasoning is parallel to the case above. \square

Next we move to transitivity of \sqsubseteq . Note that \sqsubseteq for two extensional values A_σ and B_σ is *not* set containment $A \supseteq B$, but slightly different.

Lemma E13. \sqsubseteq is transitive.

Proof. Let $e \sqsubseteq u \sqsubseteq o$ with $e, u, o \in \llbracket \sigma \rrbracket_{\mathcal{B}}$; we prove that $e \sqsubseteq o$ by induction on the form of σ . The induction is entirely straightforward:

- if $\sigma \in \mathcal{S}$, then $e = u = o$;
- if $\sigma = \sigma_1 \times \sigma_2$, then $e = (e_1, e_2)$, $u = (u_1, u_2)$ and $o = (o_1, o_2)$ with both $e_1 \sqsubseteq u_1 \sqsubseteq o_1$ and $e_2 \sqsubseteq u_2 \sqsubseteq o_2$; by the induction hypothesis indeed $e_1 \sqsubseteq o_1$ and $e_2 \sqsubseteq o_2$;
- if $\sigma = \sigma_1 \Rightarrow \sigma_2$, then we can write $e = A_\sigma$, $u = B_\sigma$ and $o = C_\sigma$ and:
 - for all $(o_1, o_2) \in C$ there exists $u_2 \sqsubseteq o_2$ such that $(o_1, u_2) \in B$;
 - for all $(o_1, u_2) \in B$ there exists $e_2 \sqsubseteq u_2$ such that $(o_1, e_2) \in A$.

As the induction hypothesis gives $e_2 \sqsubseteq o_2$, also $e \sqsubseteq o$. \square

Finally, we show how \mathbf{p} and \mathbf{p}' in Algorithm 7 relate:

Lemma E14. $\llbracket \mathbf{p} \rrbracket(d_1, \dots, d_M) \mapsto b$ if and only if $\mathbf{p}' \vdash^{\text{call}} \text{start } d_1 \dots d_M \rightarrow b$.

Proof. By Lemma 1 and the observation that the fresh symbol **start** does not occur in any other clauses, $\llbracket \mathbf{p} \rrbracket(d_1, \dots, d_M) \mapsto b$ if and only if $\llbracket \mathbf{p}' \rrbracket(d_1, \dots, d_M) \mapsto b$, which by definition is the case if and only if $\mathbf{p}', [x_1 := d_1, \dots, x_M := d_M] \vdash_{\mathbf{f}_1} x_1 \dots x_M \rightarrow b$. As there is only one clause for **start** in \mathbf{p}' , this is the case if and only if $\mathbf{p}' \vdash^{\text{call}} \text{start } d_1 \dots d_M \rightarrow b$. \square

E.2 Soundness of Algorithms 7 and 13

We turn to soundness. We will see that for every b in the output set of Algorithms 7 and 13 indeed $\llbracket \mathbf{p} \rrbracket(d_1, \dots, d_M) \mapsto b$. Since each $\langle \sigma \rangle_{\mathcal{B}} \subseteq \llbracket \sigma \rrbracket_{\mathcal{B}}$ —and therefore the statements considered in Algorithm 7 are a subset of those considered in Algorithm 13—it suffices to prove this for the non-deterministic algorithm, as the deterministic case follows directly.

To achieve this end, we first give a definition to relate values and extensional values in line with Lemma E12, and obtain two further helper results:

Definition 22. For a value $v : \sigma$ and an extensional value $e \in \llbracket \sigma \rrbracket_{\mathcal{B}}$, we recursively define $v \Downarrow e$ if one of the following holds:

- $\sigma \in \mathcal{S}$ and $v = e$;
- $\sigma = \sigma_1 \times \sigma_2$ and $v = (v_1, v_2)$ and $e = (e_1, e_2)$ with $v_1 \Downarrow e_1$ and $v_2 \Downarrow e_2$;
- $\sigma = \sigma_1 \Rightarrow \sigma_2$ and $e = A_\sigma$ with $A \subseteq \varphi(v) := \{(u_1, u_2) \mid u_1 \in \llbracket \sigma_1 \rrbracket_{\mathcal{B}} \wedge u_2 \in \llbracket \sigma_2 \rrbracket_{\mathcal{B}} \wedge \text{for all values } w_1 : \sigma_1 \text{ with } w \Downarrow u_1 \text{ there is some value } w_2 : \sigma_2 \text{ with } w_2 \Downarrow u_2 \text{ such that } \mathbf{p}' \vdash^{\text{call}} v \ w_1 \rightarrow w_2\}$.

It is easy to see that \Downarrow satisfies the requirements of Lemma E12.

The first helper lemma essentially states the following: if a value v is associated to an extensional value e (in the sense that $v \Downarrow e$), and v_1, \dots, v_n are associated to extensional values u_1, \dots, u_n , then the set $e(u_1, \dots, u_n)$ contains only (extensional values associated to) the possible results of evaluating $v \ v_1 \cdots v_n$. Thus, if $v \Downarrow e$ then e represents v in the expected sense: by defining the same “function”.

(To make it easier to use this lemma in the proof of Lemma 16, however, it is formulated in a slightly more general way than this sketch: the lemma considers an expression s which evaluates to v , and similarly expressions t_1, \dots, t_n which evaluate to each v_1, \dots, v_n . We show that $s \ t_1 \cdots t_n$ evaluates to the elements of $e(u_1, \dots, u_n)$.)

Lemma E15. *Assume given an environment γ . Let $s : \sigma_1 \Rightarrow \dots \Rightarrow \sigma_n \Rightarrow \tau$, and $e \in \llbracket \sigma_1 \Rightarrow \dots \Rightarrow \sigma_n \Rightarrow \tau \rrbracket_{\mathcal{B}}$ be such that $v \Downarrow e$ for some value v with $\mathbf{p}', \gamma \vdash s \rightarrow v$. For $1 \leq i \leq n$, let $t_i, v_i : \sigma_i$ and $u_i \in \llbracket \sigma_i \rrbracket_{\mathcal{B}}$ be such that $\mathbf{p}', \gamma \vdash t_i \rightarrow v_i \Downarrow u_i$. Then for any $o \in e(u_1, \dots, u_n)$ there exists $w : \tau$ such that $w \Downarrow o$ and $\mathbf{p}', \gamma \vdash s \ t_1 \cdots t_n \rightarrow w$.*

Proof. By induction on $n \geq 0$.

If $n = 0$, then $o = e$ and $\mathbf{p}', \gamma \vdash^{\text{call}} s \rightarrow v$ is given; we choose $w := v$.

If $n \geq 1$, then there is some $o' := A_{\sigma_n \Rightarrow \tau} \in e(u_1, \dots, u_{n-1})$ such that $(u_n, o) \in A$. By the induction hypothesis, there exists a value w' such that $\mathbf{p}', \gamma \vdash s \ t_1 \cdots t_{n-1} \rightarrow w' \Downarrow o'$. Since also $v_n \Downarrow u_n$, the definition of \Downarrow provides a value w such that $\mathbf{p}' \vdash^{\text{call}} w' \ v_n \rightarrow w \Downarrow o$. As w' is a value of higher type, it must have a form $\mathbf{f} \ w_1 \cdots w_i$, so we can apply [Appl] to obtain $\mathbf{p}', \gamma \vdash (s \ t_1 \cdots t_{n-1}) \ t_n \rightarrow w$. \square

The second helper lemma states the following: if a value v is associated to an extensional value e , then it is also associated to all “smaller” extensional values: if $e \sqsupseteq u$, then u simply has less information about the value described. The property is closely related to transitivity of \sqsupseteq :

Lemma E16. *For any value $v : \sigma$ and extensional values $e, u \in \llbracket \sigma \rrbracket_{\mathcal{B}}$: if $v \Downarrow e \sqsupseteq u$ then $v \Downarrow u$.*

Proof. By induction on the form of σ :

- if v is data, then $v = e = u$;
- if $v = (v_1, v_2)$, then $v \Downarrow e \sqsupseteq u$ implies $e = (e_1, e_2)$ and $u = (u_1, u_2)$ with $v_i \Downarrow e_i \sqsupseteq u_i$ for $i \in \{1, 2\}$, so $v_i \Downarrow u_i$ by the induction hypothesis;
- if v is a functional value, then $e = A_\sigma$ and $u = B_\sigma$, and for all $(o_1, o_2) \in B$ there exists $o'_2 \sqsupseteq o_2$ such that $(o_1, o'_2) \in A$; thus, for all values $w_1 \Downarrow o_1$, the property that $v \Downarrow e$ gives some w_2 such that $\mathbf{p}' \vdash^{\text{call}} v \ w_1 \rightarrow w_2 \Downarrow o'_2 \sqsupseteq o_2$, which by the induction hypothesis implies $w_2 \Downarrow o_2$ as well. Thus, indeed $v \Downarrow u$. \square

With these preparations, we are ready to tackle the soundness proof:

Lemma 16. *If Algorithm 7 or 13 returns a set $A \cup \{b\}$, then $\llbracket \mathbf{p} \rrbracket(d_1, \dots, d_M) \mapsto b$.*

Proof. We prove the lemma by obtaining the following results:

1. Let:
 - $\mathbf{f} : \sigma_1 \Rightarrow \dots \Rightarrow \sigma_m \Rightarrow \kappa \in \mathcal{F}$ be a defined symbol;
 - $v_1 : \sigma_1, \dots, v_n : \sigma_n$ be values, for $1 \leq n \leq \text{arity}_{\mathbf{p}}(\mathbf{f})$;
 - $e_1 \in \llbracket \sigma_1 \rrbracket_{\mathcal{B}}, \dots, e_n \in \llbracket \sigma_n \rrbracket_{\mathcal{B}}$ be such that each $v_i \Downarrow e_i$;
 - $o \in \llbracket \sigma_{n+1} \Rightarrow \dots \Rightarrow \sigma_m \Rightarrow \kappa \rrbracket_{\mathcal{B}}$.
 If the statement $\vdash \mathbf{f} e_1 \dots e_n \rightsquigarrow o$ is eventually confirmed, then we can derive $\mathbf{p}' \vdash^{\text{call}} \mathbf{f} v_1 \dots v_n \rightarrow w$ for some w with $w \Downarrow o$.
2. Let:
 - $\rho : \mathbf{f} \ell_1 \dots \ell_k = s$ be a clause in \mathbf{p}' ;
 - $t : \tau$ be a sub-expression of s ;
 - η be an ext-environment for ρ ;
 - γ be an environment such that $\gamma(x) \Downarrow \eta(x)$ for all $x \in \text{Var}(\mathbf{f} \ell_1 \dots \ell_k)$;
 - $o \in \llbracket \tau \rrbracket_{\mathcal{B}}$.
 If the statement $\eta \vdash t \rightsquigarrow o$ is eventually confirmed, then we can derive $\mathbf{p}', \gamma \vdash t \rightarrow w$ for some w with $w \Downarrow o$.

This proves the lemma: if the algorithm returns b , then $\text{start } d_1 \dots d_M \rightsquigarrow b$ is confirmed, so $\mathbf{p}' \vdash^{\text{call}} \text{start } d_1 \dots d_M \mapsto b$. By Lemma E14, $\llbracket \mathbf{p} \rrbracket(d_1, \dots, d_M) \mapsto b$.

We prove both statements together by induction on the algorithm.

1. $\mathbf{f} e_1 \dots e_n \rightsquigarrow o$ can only be confirmed in two ways:
 - (2a) $n < \text{arity}_{\mathbf{p}}(\mathbf{f})$, $o = O_{\sigma_{n+1} \Rightarrow \dots \Rightarrow \sigma_m \Rightarrow \kappa}$ and for all $(e_{n+1}, u) \in O$ there is some $u' \sqsupseteq u$ such that also $\mathbf{f} e_1 \dots e_{n+1} \rightsquigarrow u'$ is confirmed. By the induction hypothesis, this implies that for all such e_{n+1} and u' , and for all $v_{n+1} : \sigma_{n+1}$ with $v_{n+1} \Downarrow e_{n+1}$, there exists w' with $w' \Downarrow u'$ such that $\mathbf{p}' \vdash^{\text{call}} \mathbf{f} v_1 \dots v_{n+1} \rightarrow w'$. By Lemma E16, also $w' \Downarrow u$. Thus, $O \subseteq \varphi(\mathbf{f} v_1 \dots v_n)$, and $(\mathbf{f} v_1 \dots v_n) \Downarrow o$. We are done choosing $w := \mathbf{f} v_1 \dots v_n$, since $\mathbf{p}' \vdash^{\text{call}} \mathbf{f} v_1 \dots v_n \rightarrow \mathbf{f} v_1 \dots v_n$ by [Closure].
 - (2b) $n = \text{arity}_{\mathbf{p}}(\mathbf{f})$ and, for $\rho : \mathbf{f} \ell_1 \dots \ell_k = s$ the first matching clause in \mathbf{p}' and η the matching ext-environment, $\eta \vdash s \rightsquigarrow o$ is confirmed. Following Lemma E12, there exists an environment γ on domain $\text{Var}(\mathbf{f} \ell_1 \dots \ell_k)$ with each $\ell_j \gamma = v_j$ and $\gamma(x) \Downarrow \eta(x)$ for each x in the mutual domain. By the induction hypothesis, we can derive $\mathbf{p}', \gamma \vdash s \rightarrow w$ for some w with $w \Downarrow o$; by [Call] therefore $\mathbf{p}' \vdash^{\text{call}} \mathbf{f} v_1 \dots v_n \rightarrow w$ (necessarily $n = k$).
2. $\eta \vdash t \rightsquigarrow o$ can be confirmed in eight ways:
 - (1(c)i) $t \in \mathcal{V}$ and $\eta(t) \sqsupseteq o$; choosing $w = \gamma(t)$, we have $\mathbf{p}', \gamma \vdash t \rightarrow w$ by [Instance], and $w \Downarrow o$ by Lemma E16.
 - (1(c)ii) $t = \mathbf{c} t_1 \dots t_m$ with $\mathbf{c} \in \mathcal{C}$ and $t\eta = o$; choosing $w = t\gamma = o$, we clearly have $w \Downarrow o$ and $\mathbf{p}', \gamma \vdash t \rightarrow w$ by [Constructor].
 - (2c) $t = \text{if } t_1 \text{ then } t_2 \text{ else } t_3$ and either
 - (2(c)i) $\eta \vdash t_1 \rightsquigarrow \text{true}$ and $\eta \vdash t_2 \rightsquigarrow o$ are both confirmed; by the induction hypothesis, $\mathbf{p}', \gamma \vdash t_1 \rightarrow \text{true}$ and $\mathbf{p}', \gamma \vdash t_2 \rightarrow w$ for some w with $w \Downarrow o$;
 - (2(c)ii) $\eta \vdash t_1 \rightsquigarrow \text{false}$ and $\eta \vdash t_3 \rightsquigarrow o$ are both confirmed; by the induction hypothesis, $\mathbf{p}', \gamma \vdash t_1 \rightarrow \text{true}$ and $\mathbf{p}', \gamma \vdash t_3 \rightarrow w$ for some w with $w \Downarrow o$.

In either case we complete with [Conditional], using [Cond-True] in the former and [Cond-False] in the latter case.

- (2d) $t = \text{choose } t_1 \cdots t_n$ and $\eta \vdash t_i \rightarrow o$ is confirmed for some i ; by the induction hypothesis, $\mathbf{p}', \gamma \vdash t_i \rightarrow w$ for a suitable w , so $\mathbf{p}', \gamma \vdash t \rightarrow w$ by [Choice].
- (2e) $t = (t_1, t_2)$ and $o = (o_1, o_2)$ and $\eta \vdash t_i \rightarrow o_i$ is confirmed for $i \in \{1, 2\}$; by the induction hypothesis, $\mathbf{p}', \gamma \vdash t_i \rightarrow w_i \Downarrow o_i$ for both i , so $\mathbf{p}', \gamma \vdash t \rightarrow (w_1, w_2) \Downarrow o$ by [Pair].
- (2f) $t = x \ t_1 \cdots t_n$ with $x \in \mathcal{V}$ and $n > 0$, and there are e_1, \dots, e_n such that $\eta \vdash t_i \rightsquigarrow e_i$ is confirmed for all i , and $\eta(x)(e_1, \dots, e_n) \ni o' \sqsupseteq o$ for some o' ; by the induction hypothesis, there are v_1, \dots, v_n such that $\mathbf{p}', \gamma \vdash t_i \rightarrow v_i$ for all i . Since also $\mathbf{p}', \gamma \vdash x \rightarrow \gamma(x) \Downarrow \eta(x)$ by [Instance], Lemma E15 provides w such that $\mathbf{p}', \gamma \vdash x \ t_1 \cdots t_n \rightarrow w \Downarrow o'$; by Lemma E16, also $w \Downarrow o$.
- (2g)i) $t = \mathbf{f} \ t_1 \cdots t_n$ with $\mathbf{f} \in \mathcal{D}$ and $0 \leq n \leq \text{arity}_{\mathbf{p}}(\mathbf{f})$, and there are e_1, \dots, e_n such that $\eta \vdash t_i \rightsquigarrow e_i$ is confirmed for all i , and $\vdash \mathbf{f} \ e_1 \cdots e_n \rightsquigarrow o$ is marked confirmed. By the second induction hypothesis, there are v_1, \dots, v_n such that $\mathbf{p}', \gamma \vdash t_i \rightarrow v_i \Downarrow e_i$ for all i , and therefore by the first induction hypothesis, there is w such that $\mathbf{p}' \vdash^{\text{call}} \mathbf{f} \ v_1 \cdots v_n \rightarrow w \Downarrow o$. Combining this with [Function] and n [Appl]s, we have $\mathbf{p}', \gamma \vdash \mathbf{f} \ t_1 \cdots t_n \rightarrow w$ as well.
- (2g)ii) $t = \mathbf{f} \ t_1 \cdots t_n$ with $\mathbf{f} \in \mathcal{D}$ and $n > k := \text{arity}_{\mathbf{p}}(\mathbf{f})$, and there are e_1, \dots, e_n such that, just as in the previous two cases, $\mathbf{p}', \gamma \vdash t_i \rightarrow v_i \Downarrow e_i$ for each i . Moreover, $u(e_{k+1}, \dots, e_n) \ni o' \sqsupseteq o$ for some u with $\mathbf{f} \ e_1 \cdots e_k \rightsquigarrow u$ confirmed. As in the previous case, there exists v such that $\mathbf{p}', \gamma \vdash \mathbf{f} \ t_1 \cdots t_k \rightarrow v \Downarrow u$. Lemma E15 provides w with $\mathbf{p}', \gamma \vdash \mathbf{f} \ t_1 \cdots t_n \rightarrow w \Downarrow o'$; since $o' \sqsupseteq o$ also $w \Downarrow o$ by Lemma E16. \square

E.3 Completeness of Algorithm 13

We turn to completeness; in particular, the property that if $\llbracket \mathbf{p} \rrbracket(d_1, \dots, d_M) \mapsto b$ then Algorithm 13 returns a set containing b (in Section E.4 we will see that for deterministic programs also Algorithm 7 returns such a set). We will do this by induction on the derivation tree; specifically, by going from the tree *right-to-left*, *top-to-bottom*. To make this induction formal, we will need to *label* the nodes; to obtain the desired order of traversing the nodes, we label them with strings, ordered in reverse lexicographic order.

Definition 23. *For a given derivation tree T , we label the nodes by strings of numbers as follows: the root is labelled 0, and for a tree*

$$\frac{T_1 \quad \dots \quad T_n}{\pi}$$

if node π is labelled with l , then we label each T_i with $l \cdot i$.

We say that $l > p$ if l is larger than p in the lexicographic ordering (with $l \cdot i > l$), and $l \succ p$ if $l > p$ but p is not a prefix of l .

Thus, for nodes labelled l and p , we have $l \succ p$ if l occurs to the right of p , and $l > p$ if l occurs to the right or above of p . We have $1 \succ l$ for all l in the tree.

In the soundness proof (Lemma 16), we essentially recursed over the steps in the algorithm, and associated to every extensional value an expression value. Now, we must go in the other direction, and associate to every value an extensional value. If a value occurs at multiple places in the derivation tree, we do not need to select the same extensional value every time—just as the soundness proof did not always associate the same value to a given extensional value.

In order to choose a suitable extensional value for each position in the derivation tree, we define the function ψ which considers the tree above and to the right of a given position. As a result, functional values are associated to ever larger extensional values as we traverse the tree right-to-left, top-to-bottom.

Definition 24. Let T be a derivation tree and L the set of its labels, which must all have the form $0 \cdot l$. For any $v \in \text{Value}_{d_1, \dots, d_M}^p$ (see Definition 21) and $l \in L \cup \{1\}$, let:

- $\psi(v, l) = v$ if $v \in \mathcal{B}$
- $\psi(v, l) = (\psi(v_1, l), \psi(v_2, l))$ if $v = (v_1, v_2)$
- for $\mathbf{f} \ v_1 \cdots v_n : \tau = \sigma_{n+1} \Rightarrow \dots \Rightarrow \sigma_m \Rightarrow \kappa$ with $m > n$, let $\psi(\mathbf{f} \ v_1 \cdots v_n, l) = \{(e_{n+1}, u) \mid \exists q \succ p > l \text{ [the subtree with index } p \text{ has a root } \mathbf{p}' \vdash^{\text{call}} \mathbf{f} \ v_1 \cdots v_{n+1} \rightarrow w \text{ with } \psi(w, q) = u \text{ and } e_{n+1} \sqsupseteq' \psi(v_{n+1}, p)]\}_\tau$. In this, q is allowed to be 1 (but p is not).

Here, \sqsupseteq' is defined the same as \sqsupseteq , except that $A_\sigma \sqsupseteq' B_\sigma$ iff $A \supseteq B$. Note that clearly $e \sqsupseteq' u$ implies $e \sqsupseteq u$, and that \sqsupseteq' is transitive by transitivity of \sqsupseteq .

Thus, $\psi(v, l) \in \llbracket \sigma \rrbracket_{\mathcal{B}}$ for $v : \sigma$, but not $\psi(v, l) \in \langle \sigma \rangle_{\mathcal{B}}$. Note that $\psi(v, l) \sqsupseteq' \psi(v, p)$ if $p > l$ by transitivity of $>$. Note also that, in the derivation tree for $\mathbf{p}' \vdash^{\text{call}} \text{start } d_1 \cdots d_M \rightarrow b$, all values are in $\text{Value}_{d_1, \dots, d_M}^p$ as all d_i are in \mathcal{B} .

Remark 5. Some choices in Definition 24 may well confound the reader.

First, the special label 1 is used because we will make statements of the form “for all $p \succ l$ there exists $o \sqsupseteq' \psi(w, p)$ with property P”: if we did not include 1 in this quantification, it would give no information about, e.g., the root of the tree. Note that this is already used in the definition of ψ .

Second, one may wonder why we use \sqsupseteq' rather than \sqsupseteq . This is purely for the sake of the proof: the simpler and more restrictive relation \sqsupseteq' works better in the induction because whenever $A_\sigma \sqsupseteq' B_\sigma$, all elements of B are also in A . In fact, in Algorithm 13 we could replace all uses of \sqsupseteq by \sqsupseteq' without affecting the algorithm’s correctness. However, this *would* be problematic for Algorithm 7; in the completeness proof in Section E.4, we will for instance use that $\{(e, u)\}_\sigma \sqsupseteq \{(e, u), (e, o)\}_\sigma$ if $u \sqsupseteq o$, something which does not hold for \sqsupseteq' .

Now, we could easily follow the proof sketch in the running text and prove directly that Algorithm 13 is complete, by showing for each subtree with a label l and root $\mathbf{p}' \vdash^{\text{call}} \mathbf{f} \ v_1 \cdots v_n \rightarrow w$ that for all e_1, \dots, e_n such that each

$e_i \sqsubseteq' \psi(v_i, l)$ and for all $p \succ l$ there exists $o \sqsubseteq' \psi(w, p)$ such that $\vdash \mathbf{f} \ e_1 \cdots e_n \rightsquigarrow o$ is eventually confirmed (and similar for subtrees $\mathbf{p}', \gamma \vdash s \rightarrow w$). However, the proof for this is quite long, and we would have to essentially repeat it with some minor changes when proving completeness of Algorithm 7.

Instead, we will take a slight detour. We present a new set of derivation rules built on extensional values, which directly corresponds to the algorithm. By these derivation rules—as presented in Figure 10—it is easy to see that $\mathbf{p}' \Vdash^{\text{call}} \mathbf{f} \ e_1 \cdots e_k \Rightarrow o$ if and only if $\vdash \mathbf{f} \ e_1 \cdots e_k \rightsquigarrow o$ is eventually confirmed in Algorithm 13, and $\mathbf{p}', \eta \Vdash^{\text{call}} s \Rightarrow o$ if and only if $\eta \vdash s \rightsquigarrow o$ is eventually confirmed. Thus, the primary work is in showing that such a derivation exists.

$$\begin{array}{l}
\text{[Constructor]} \quad \frac{}{\mathbf{p}', \eta \Vdash \mathbf{c} \ s_1 \cdots s_m \Rightarrow \mathbf{c} \ (s_1 \eta) \cdots (s_m \eta)} \\
\text{[Pair]} \quad \frac{\mathbf{p}', \eta \Vdash s \Rightarrow o_1 \quad \mathbf{p}', \eta \Vdash t \Rightarrow o_2}{\mathbf{p}', \eta \Vdash (s, t) \Rightarrow (o_1, o_2)} \\
\text{[Choice]} \quad \frac{\mathbf{p}', \eta \Vdash s_i \Rightarrow o}{\mathbf{p}', \eta \Vdash \text{choose } s_1 \cdots s_n \Rightarrow o} \text{ for } 1 \leq i \leq n \\
\text{[Cond-True]} \quad \frac{\mathbf{p}', \eta \Vdash s_1 \Rightarrow \text{true} \quad \mathbf{p}', \eta \Vdash s_2 \Rightarrow o}{\mathbf{p}', \eta \Vdash \text{if } s_1 \text{ then } s_2 \text{ else } s_3 \Rightarrow o} \\
\text{[Cond-False]} \quad \frac{\mathbf{p}', \eta \Vdash s_1 \Rightarrow \text{false} \quad \mathbf{p}', \eta \Vdash s_3 \Rightarrow o}{\mathbf{p}', \eta \Vdash \text{if } s_1 \text{ then } s_2 \text{ else } s_3 \Rightarrow o} \\
\text{[Variable]} \quad \frac{\mathbf{p}', \eta \Vdash s_i \Rightarrow e_i \text{ for } 1 \leq i \leq n}{\mathbf{p}', \eta \Vdash x \ s_1 \cdots s_n \Rightarrow o} \exists o' \in \eta(x)(e_1, \dots, e_n)[o' \sqsupseteq o] \\
\text{[Func]} \quad \frac{\mathbf{p}', \eta \Vdash s_i \Rightarrow e_i \text{ for } 1 \leq i \leq n \quad \mathbf{p}' \Vdash^{\text{call}} \mathbf{f} \ e_1 \cdots e_n \Rightarrow o}{\mathbf{p}', \eta \Vdash \mathbf{f} \ s_1 \cdots s_n \Rightarrow o} \text{ for } \mathbf{f} \in \mathcal{D}, \ n \leq \text{arity}_{\mathbf{p}}(\mathbf{f}) \\
\text{[Applied]} \quad \frac{\mathbf{p}', \eta \Vdash s_i \Rightarrow e_i \text{ for } 1 \leq i \leq n \quad \mathbf{p}' \Vdash^{\text{call}} \mathbf{f} \ e_1 \cdots e_k \Rightarrow u}{\mathbf{p}', \eta \Vdash \mathbf{f} \ s_1 \cdots s_n \Rightarrow o} \text{ for } \mathbf{f} \in \mathcal{D}, \ n > \text{arity}_{\mathbf{p}}(\mathbf{f}), \ o' \in u(e_{k+1}, \dots, e_n), \ o' \sqsupseteq o \\
\text{[Value]} \quad \frac{\mathbf{p}' \Vdash^{\text{call}} \mathbf{f} \ e_1 \cdots e_{n+1} \Rightarrow u' \sqsupseteq u \text{ for all } (e_{n+1}, u) \in O}{\mathbf{p}' \Vdash^{\text{call}} \mathbf{f} \ e_1 \cdots e_n \Rightarrow O_{\sigma}} \text{ if } n < \text{arity}_{\mathbf{p}}(\mathbf{f}) \\
\text{[Call]} \quad \frac{\mathbf{p}', \eta \Vdash s \Rightarrow o}{\mathbf{p}' \Vdash^{\text{call}} \mathbf{f} \ e_1 \cdots e_k \Rightarrow o} \text{ if } \mathbf{f} \ \ell_1 \cdots \ell_k = s \text{ is the first clause in } \mathbf{p}' \text{ which matches } \mathbf{f} \ e_1 \cdots e_k, \text{ and } \eta \text{ is the matching ext-environment}
\end{array}$$

Fig. 10. Alternative semantics using (non-deterministic) extensional values

Thus, we come to the main result needed for completeness:

Lemma E17. *If $\llbracket \mathbf{p} \rrbracket(d_1, \dots, d_M) \mapsto b$, then $\mathbf{p}' \Vdash^{\text{call}} \text{start } d_1 \cdots d_M \Rightarrow b$.*

Proof. Given $\llbracket p \rrbracket(d_1, \dots, d_M) \mapsto b$, Lemma E14 allows us to assume that $p' \vdash^{\text{call}} \text{start } d_1 \cdots d_M \rightarrow b$. Let T be the derivation tree with this root (with root label 0) and L the set of its labels. We prove, by induction on l with greater labels handled first (which is well-founded because T has only finitely many subtrees):

1. If the subtree with label l has root $p' \vdash^{\text{call}} \mathbf{f } v_1 \cdots v_n \rightarrow w$, then for all e_1, \dots, e_n such that each $e_i \sqsubseteq' \psi(v_i, l)$, and for all $p \succ l$ there exists $o \sqsubseteq' \psi(w, p)$ such that $p' \Vdash^{\text{call}} \mathbf{f } e_1 \cdots e_n \Rightarrow o$.
2. If the subtree with label l has root $p', \gamma \vdash t \rightarrow w$ and $\eta(x) \sqsubseteq' \psi(\gamma(x), l)$ for all $x \in \text{Var}(t)$, then for all $p \succ l$ there exists $o \sqsubseteq' \psi(w, p)$ such that $p', \eta \Vdash t \Rightarrow o$.

Here, for $p \succ l$ we allow $p \in L \cup \{1\}$. Therefore, in both cases, there must exist a suitable $o \sqsubseteq' \psi(w, 1)$ if w is a data expression; this o can only be w itself. The first item gives the desired result for $l = 0$, as $o \sqsubseteq' \psi(b, 1)$ implies $o = b$.

We prove both items together by induction on l , with greater labels handled first. Consider the first item. There are two cases:

- If $p' \vdash^{\text{call}} \mathbf{f } v_1 \cdots v_n \rightarrow w$ by [Closure], then $n < \text{arity}_p(\mathbf{f})$ and $w = \mathbf{f } v_1 \cdots v_n$. Given $p \succ l$, let $o := \psi(w, l)$; then clearly $o \sqsubseteq' \psi(w, p)$. We must see that $p' \Vdash^{\text{call}} \mathbf{f } e_1 \cdots e_n \Rightarrow o$; by [Value], this is the case if for all (e_{n+1}, u) in the set underlying o we can derive $p' \Vdash^{\text{call}} \mathbf{f } e_1 \cdots e_{n+1} \Rightarrow u'$ for some $u' \sqsubseteq' u$. So let (e_{n+1}, u) be in this underlying set. By definition of ψ , we can find $q \succ p' > l$ and v_{n+1}, w' such that the subtree with label p' has a root $p' \vdash^{\text{call}} \mathbf{f } v_1 \cdots v_{n+1} \rightarrow w'$ and $e_{n+1} \sqsubseteq' \psi(v_{n+1}, p')$ and $u = \psi(w', q)$. Since $p' > l$, also $e_i \sqsubseteq' \psi(v_i, p')$ for $1 \leq i \leq n$; thus, the induction hypothesis provides $u' \sqsubseteq' \psi(w', q) = u$ with $p' \vdash^{\text{call}} \mathbf{f } e_1 \cdots e_{n+1} \Rightarrow u'$ as required.
- If $p' \vdash^{\text{call}} \mathbf{f } v_1 \cdots v_n \rightarrow w$ by [Call], then $n = \text{arity}_p(\mathbf{f})$ and we can find a clause, say $\rho: \mathbf{f } \ell_1 \cdots \ell_n = s$ and an environment γ such that
 1. ρ is the first clause in p' whose right-hand side is instantiated by $\mathbf{f } v_1 \cdots v_n$;
 2. each $v_i = \ell_i \gamma$;
 3. $p', \gamma \vdash s \rightarrow w$.
 By Lemma E12, using $v \Downarrow V$ iff $V \sqsubseteq' \psi(v, l)$, also ρ is the first clause which matches $\mathbf{f } e_1 \cdots e_n$, and for the matching ext-environment η , each $\eta(x) \sqsubseteq' \psi(\gamma(x), l) \sqsubseteq' \psi(\gamma(x), l \cdot 1)$. Thus using the induction hypothesis for observation 3, we find $o \sqsubseteq' \psi(w, p)$ for all $p \succ l \cdot 1$. As this includes every label p with $p \succ l$, we are done.

Now for the second claim, assume that $p', \gamma \vdash t \rightarrow w$ (with label l) and that $\eta(x) \sqsubseteq' \psi(\gamma(x), l)$ for all $x \in \text{Var}(t)$; let $p \succ l$ which (**) implies $p \succ l \cdot i$ for any string i as well. Consider the form of t (taking into account that, following the transformation of p to p' , we do not need to consider applications whose head is an **if then else** or **choose** statement).

- $t = (t_1, t_2)$; then we can write $w = (w_1, w_2)$ and the trees with labels $l \cdot 1$ and $l \cdot 2$ have roots $p', \gamma \vdash t_1 \rightarrow w_1$ and $p', \gamma \vdash t_2 \rightarrow w_2$ respectively. Using observation (**), the induction hypothesis provides o_1, o_2 such that each $p', \eta \Vdash s_i \Rightarrow o_i \sqsubseteq' \psi(w_i, p)$; we are done choosing $o := (o_1, o_2)$.

- $t = c \, t_1 \cdots t_m$ with $c \in \mathcal{C}$; then by Lemma B7, each $s_i \gamma = b_i \in \mathcal{B}$; this implies that all $\gamma(x) \in \mathcal{B}$, so $\eta(x) = \gamma(x)$, and $\mathbf{p}', \eta \vdash t \Rightarrow o := t\eta$ by [Constructor].
- $t = \text{choose } t_1 \cdots t_n$; then the immediate subtree is $\mathbf{p}', \gamma \vdash t_i \rightarrow w$ for some i . By observation (**), the induction hypothesis provides a suitable o , which suffices by rule [Choice] from \Vdash .
- $t = \text{if } t_1 \text{ then } t_2 \text{ else } t_3$; then, as the immediate subtree can only be obtained by [If-True] or [If-False], we have either $\mathbf{p}', \gamma \vdash s_1 \rightarrow \text{true}$ and $\mathbf{p}', \gamma \vdash s_2 \rightarrow w$, or $\mathbf{p}', \gamma \vdash s_1 \rightarrow \text{false}$ and $\mathbf{p}', \gamma \vdash s_3 \rightarrow w$. Using the induction hypothesis for $p = 1$, we have $\mathbf{p}', \eta \Rightarrow \text{true}$ in the first case and $\mathbf{p}', \eta \Rightarrow \text{false}$ in the second. Using (**) and the induction hypothesis as before, we obtain a suitable o using the inference rule [Cond-True] or [Cond-False] of \Vdash .
- $t \in \mathcal{V}$, so the tree is obtained by [Instance]; choosing $o := \eta(t) \sqsupseteq' \psi(\gamma(t), l) \sqsupseteq' \psi(\gamma(t), p)$ by (**), we have $o \sqsupseteq' \psi(w, p)$ by transitivity of \sqsupseteq' , and $\mathbf{p}', \eta \Vdash t \Rightarrow o$ by [Variable] (as $o \in \{o\} = o()$).
- $t = x \, t_1 \cdots t_n$ with $n > 0$; then there are w_0, \dots, w_n such that the root is obtained using:
 - $\mathbf{p}', \gamma \vdash x \rightarrow \gamma(x) =: w_0$ by [Instance] with label $l \cdot 1^n$;
 - n subtrees of the form $\mathbf{p}', \gamma \vdash t_i \rightarrow v_i$ with label $l \cdot 1^{n-i} \cdot 2$ for $1 \leq i \leq n$;
 - n subtrees of the form $\mathbf{p}' \vdash^{\text{call}} w_{i-1} \, v_i \rightarrow w_i$ with label $l \cdot 1^{n-i} \cdot 3$ for $1 \leq i \leq n$;
 - n uses of [Appl], each with conclusion w_i and label $l \cdot 1^{n-i}$ for $1 \leq i \leq n$.
 Note that here $w_n = w$. For $1 \leq i \leq n$ we define e_i and o_{i-1} as follows:
 - observing that $l \cdot 1^{n-i} \cdot 3 \succ l \cdot 1^{n-i} \cdot 2$, the induction hypothesis provides e_i such that $\mathbf{p}', \eta \vdash t_i \Rightarrow e_i \sqsupseteq' \psi(v_i, l \cdot 1^{n-i} \cdot 3)$
 - $o_0 := \psi(\gamma(x), l) \sqsupseteq' \psi(w_i, l \cdot 1^{n-1} \cdot 2)$;
 - for $1 < i \leq n$, let $o_{i-1} := \psi(w_{i-1}, l \cdot 1^{n-i} \cdot 2)$.
 We also define $o_n := \psi(w_n, p)$. Then by definition of ψ , because $l \cdot 1^{n-i} \cdot 3 > l \cdot 1^{n-i} \cdot 2$ and the former is the label of $\mathbf{p}' \vdash^{\text{call}} w_{i-1} \, v_i \rightarrow w_i$, there is an element $(e_i, \psi(w_i, q))$ in the set underlying o_{i-1} for any $q \succ l \cdot 1^{n-i} \cdot 3$. In particular, this means (e_i, o_i) is in this set, whether $i < n$ or $i = n$. Thus, by a quick induction on i we have $o_i \in \eta(x)(e_1, \dots, e_i)$, so $\mathbf{p}', \eta \Vdash s \Rightarrow o_n = \psi(w, p)$ by [Variable].
- $t = \mathbf{f} \, t_1 \cdots t_n$ with $n \leq \text{arity}_{\mathbf{p}}(\mathbf{f})$; then there are subtrees $\mathbf{p}', \gamma \vdash t_i \rightarrow v_i$ labelled $l \cdot 1^{n-i} \cdot 2$ and $\mathbf{p}' \vdash \mathbf{f} \, v_1 \cdots v_n \rightarrow w$ labelled $l \cdot 3$. By the induction hypothesis, there are e_1, \dots, e_n such that $\mathbf{p}', \eta \vdash t_i \Rightarrow e_i \sqsupseteq' \psi(v_i, l \cdot 3)$ for $1 \leq i \leq n$. Therefore, by the \vdash^{call} part of the induction hypothesis and (**), there is $o \sqsupseteq' \psi(w, p)$ such that $\mathbf{p}', \eta \Vdash \mathbf{f} \, e_1 \cdots e_n \Rightarrow o$. But then $\mathbf{p}', \eta \Vdash t \Rightarrow o$ by [Func].
- $t = (\mathbf{f} \, s_1 \cdots s_k) \, t_1 \cdots t_0$ with $k = \text{arity}_{\mathbf{p}}(\mathbf{f})$ and $n > 0$; then there are subtrees:
 - $\mathbf{p}', \gamma \vdash \mathbf{f} \, s_1 \cdots s_k \rightarrow w_0$ by [Function] or [Appl], with label $l \cdot 1^n$;
 - $\mathbf{p}', \gamma \vdash t_i \rightarrow v_i$ with label $l \cdot 1^{n-i} \cdot 2$ for $1 \leq i \leq n$;
 - $\mathbf{p}' \vdash^{\text{call}} w_{i-1} \, v_i \rightarrow w_i$ with label $l \cdot 1^{n-i} \cdot 3$ for $1 \leq i \leq n$.

For some w_0, \dots, w_n with $w_n = w$. In the same way as the previous case, there exists $o_0 \sqsupseteq' \psi(w_0, l \cdot 1^{n-1} \cdot 2)$ such that $\mathbf{p}', \eta \Vdash \mathbf{f} \, s_1 \cdots s_k \Rightarrow o_0$ (as $l \cdot 1^{n-1} \cdot 2$). The remainder of this case follows the case with $t = x \, t_1 \cdots t_n$. \square

Now we may forget ψ and \sqsubseteq' altogether: from a derivation for $\llbracket p \rrbracket(d_1, \dots, d_M) \mapsto b$ we have obtained a derivation $p' \Vdash \text{start } d_1 \cdots d_M \Rightarrow b$, which almost exactly corresponds to a derivation of $\vdash \text{start } d_1 \cdots d_M \leadsto b$ using Algorithm 13. It only remains to formalise this correspondence:

Lemma E18. *If $p' \Vdash^{\text{call}} \text{start } d_1 \cdots d_M \Rightarrow b$, then Algorithm 13 returns a set containing b . If $p' \Vdash^{\text{call}} \text{start } d_1 \cdots d_M \Rightarrow b$ has a derivation tree which only uses deterministic extensional values, then so does Algorithm 7.*

Proof. This is entirely straightforward. Starting with $\mathcal{B} = \mathcal{B}_{d_1, \dots, d_M}^p$, we show:

1. If $p' \Vdash^{\text{call}} f \ e_1 \cdots e_n \Rightarrow o$, then $\vdash f \ e_1 \cdots e_n \leadsto o$ is eventually confirmed.
2. If $p', \eta \Vdash s \Rightarrow o$, then $\eta \vdash s \leadsto o$ is eventually confirmed.

Both statements hold regardless of which algorithm is used, provided that all extensional values in the derivation tree are among those considered by the algorithm. We prove the statements together by induction on the derivation tree. For the first, there are two inference rules that might have been used:

Value $o = O_\sigma$ and for all $(e_{n+1}, u) \in O$ there exists $u' \sqsupseteq u$ such that $p' \Vdash^{\text{call}} f \ e_1 \cdots e_{n+1} \Rightarrow u'$ is an immediate subtree. By the induction hypothesis, each such statement $f \ e_1 \cdots e_{n+1} \leadsto u'$ is confirmed, so the current statement is confirmed by step 2a.

Call Immediate by the induction hypothesis and step 2b.

For the second, suppose $p', \eta \Vdash s \Rightarrow o$, and consider the inference rule used to derive this.

Constructor Immediate by step 1c.

Pair Immediate by the induction hypothesis and step 2e.

Choice Immediate by the induction hypothesis and step 2d.

Cond-True Immediate by the induction hypothesis and step 2c.

Variable If $n = 0$, then $\eta(x) \sqsupseteq o$, so the statement is confirmed in step 1c. Otherwise, by the induction hypothesis $\eta \vdash s_i \leadsto e_i$ is confirmed for each i and $o' \sqsupseteq o$ for some $o' \in \eta(x)(e_1, \dots, e_n)$; the statement is confirmed in step 2f.

Func Immediate by the induction hypothesis and step 2(g)i.

Applied Immediate by the induction hypothesis and step 2(g)ii. \square

At this point, we have all the components for Lemma 17.

Lemma 17. *If $\llbracket p \rrbracket(d_1, \dots, d_M) \mapsto b$, then Algorithm 13 returns a set $A \cup \{b\}$.*

Proof. Immediate by a combination of Lemmas E17 and E18. \square

E.4 Completeness of Algorithm 7

Now we turn to the deterministic case. By Lemma E18, it suffices if we can find a derivation of $\mathbf{p}' \Vdash^{\text{call}} \text{start } d_1 \cdots d_M \Rightarrow b$ which uses only deterministic extensional values (elements of some $\langle \sigma \rangle_{\mathcal{B}}$). While the tree that we built in Lemma E17 does not have this property, we will use it to build a tree which does.

To start, we will see that the conclusions in any derivation tree are consistent, where *consistency* of two extensional values is defined as follows:

- $d \wr b$ iff $d = b$ for $d, b \in \text{Data}$;
- $(e_1, u_1) \wr (e_2, u_2)$ iff both $e_1 \wr e_2$ and $u_1 \wr u_2$;
- $A_\sigma \wr B_\sigma$ iff for all $(e_1, u_1) \in A$ and $(e_2, u_2) \in B$: if $e_1 \wr e_2$ then $u_1 \wr u_2$.

Consistency is preserved under taking “smaller” extensional values:

Lemma E19. *If $e'_1 \sqsupseteq e_1$, $e'_2 \sqsupseteq e_2$ and $e'_1 \wr e'_2$, then also $e_1 \wr e_2$.*

Proof. By induction on the form of e_1 . If $e_1 \in \mathcal{B}$, then $e'_1 = e_1 = e_2 = e'_2$. If e_1 is a pair, then so is e'_1 and we use the induction hypothesis. Finally, suppose $e_1 = B_\sigma^1, e_2 = B_\sigma^2, e'_1 = A_\sigma^1$ and $e'_2 = A_\sigma^2$. Then for all $(u_1, o_1) \in B^1$ and $(u_2, o_2) \in B^2$, there are $o'_1 \sqsupseteq o_1$ and $o'_2 \sqsupseteq o_2$ such that $(u_1, o'_1) \in A^1$ and $(u_2, o'_2) \in A^2$. Now suppose $u_1 \wr u_2$. By consistency of e_1 and e_2 , we then have $o'_1 \wr o'_2$, so by the induction hypothesis, also $o_1 \wr o_2$. This gives $B_\sigma^1 \wr B_\sigma^2$, so $e_1 \wr e_2$. \square

We can use this to see that conclusions using \Rightarrow are consistent; that is, if all extensional values on the left of \Rightarrow are consistent, then so are those on the right:

Lemma E20. *Let T_1, T_2 be derivation trees for \Vdash , and let $\text{root}(T_1), \text{root}(T_2)$ denote their roots. Suppose given o, o' such that one of the following holds:*

1. *There are $\mathbf{f}, e_1, \dots, e_n, e'_1, \dots, e'_n$ such that:*
 - $\text{root}(T_1) = \mathbf{f} \ e_1 \cdots e_n \Rightarrow o$;
 - $\text{root}(T_2) = \mathbf{f} \ e'_1 \cdots e'_n \Rightarrow o'$;
 - $e_1 \wr e'_1, \dots, e_n \wr e'_n$.
2. *There are η, η' on the same domain and s such that:*
 - $\text{root}(T_1) = \mathbf{p}', \eta \Vdash s \Rightarrow o$;
 - $\text{root}(T_2) = \mathbf{p}', \eta' \Vdash s \Rightarrow o'$;
 - $\eta(x) \wr \eta'(x)$ for all x occurring in s .

Moreover, s has no sub-expressions of the form $(\text{if } b \text{ then } s_1 \text{ else } s_2) \ t_1 \cdots t_n$ with $n > 0$.

If choose does not occur in s or any clause of \mathbf{p}' , then $o \wr o'$.

Proof. Both statements are proved together by induction on the form of T_1 . For the first, consider n . Since $\text{root}(T_1)$ could be derived, necessarily $n \leq \text{arity}_{\mathbf{p}}(\mathbf{f})$. There are two cases:

- $n < \text{arity}_{\mathbf{p}}(\mathbf{f})$; both trees were derived by [Value]. Thus, we can write $o = A_\sigma$ and $o' = A'_\sigma$ and have:

- for all $(e_{n+1}, u_1) \in A$ there is some $u_2 \sqsupseteq u_1$ such that T_1 has an immediate subtree $\mathbf{p}' \Vdash \mathbf{f} \ e_1 \cdots e_n \ e_{n+1} \Rightarrow u_2$;
- for all $(e'_{n+1}, u'_1) \in A'$ there is some $u'_2 \sqsupseteq u'_1$ such that T_2 has an immediate subtree $\mathbf{p}' \Vdash \mathbf{f} \ e'_1 \cdots e'_n \ e'_{n+1} \Rightarrow u'_2$.

Now let $(e_{n+1}, u_1) \in A$ and $(e'_{n+1}, u'_1) \in B$ be such that $e_{n+1} \wr e'_{n+1}$. Considering the two relevant subtrees, the induction hypothesis gives that $u_2 \wr u'_2$.

By Lemma E19 we then obtain the required property that $u_1 \wr u'_1$.

- $n = \text{arity}_{\mathbf{p}}(\mathbf{f})$; both trees were derived by [Call]. Given that extensional values of the form A_σ can only instantiate variables (not pairs or patterns with a constructor at the head), a reasoning much like the one in Lemma E12 gives us that both conclusions are obtained by the same clause $\mathbf{f} \ \ell_1 \cdots \ell_k = s$, the first with ext-environment η and the second with η' such that each $\eta(x) \wr \eta'(x)$. Then the immediate subtrees have roots $\mathbf{p}', \eta \Vdash s \Rightarrow o$ for T_1 and $\mathbf{p}', \eta' \Vdash s \Rightarrow o'$ for T_2 , and we are done by the induction hypothesis.

For the second statement, let T_1 have a root $\eta \Vdash s \Rightarrow o$ and T_2 a root $\eta' \Vdash s \Rightarrow o'$, and assume that s does not contain any **choose** operators or if-statements at the head of an application. In addition, let $\eta(x) \wr \eta'(x)$ for all (relevant) x . Then s may have one of six forms:

- $s = \mathbf{c} \ s_1 \cdots s_m$: then $o = s\eta$ and $o' = s\eta'$; as, in this case, necessarily all variables have a type of order 0, $o = o'$ which guarantees consistency.
- $s = (s_1, s_2)$: then $o = (o_1, o_2)$ and $o' = (o'_1, o'_2)$, and by the induction hypothesis both $o_1 \wr o'_1$ and $o_2 \wr o'_2$; thus indeed $o \wr o'$.
- $s = \text{if } s_1 \text{ then } s_2 \text{ else } s_3$: since not **true** \wr **false**, either both conclusions are derived by [Cond-True] or by [Cond-False]; consistency of o and o' follows immediately by the induction hypothesis on the second subtree.
- $s = x \ s_1 \cdots s_n$: the induction hypothesis provides e_1, \dots, e_n and e'_1, \dots, e'_n such that each $e_i \wr e'_i$ and there are $u \sqsupseteq o, u' \sqsupseteq o'$ such that $u \in \eta(x)(e_1, \dots, e_n)$ and $u' \in \eta'(x)(e'_1, \dots, e'_n)$. By Lemma E19, it suffices if u and u' are consistent.

We prove this by induction on n :

- if $n = 0$ then $u = \eta(x)$ and $u' = \eta'(x)$ and consistency is assumed;
- if $n > 0$ then there are $A_\sigma \in \eta(x)(e_1, \dots, e_{n-1})$ and $B_\sigma \in \eta'(x)(e'_1, \dots, e'_{n-1})$ such that $(e_n, u) \in A$ and $(e'_n, u') \in B$. By the induction hypothesis, $A_\sigma \wr B_\sigma$. Since also $e_n \wr e'_n$, this implies $u \wr u'$.
- $s = \mathbf{f} \ s_1 \cdots s_n$ with $n \leq \text{arity}_{\mathbf{p}}(\mathbf{f})$; then both conclusions follow by [Func]. The immediate subtrees provide e_1, \dots, e_n and e'_1, \dots, e'_n such that, by the induction hypothesis, each $e_i \wr e'_i$, as well as a conclusion $\mathbf{p}' \Vdash^{\text{call}} \mathbf{f} \ e_1 \cdots e_n \Rightarrow o$ in T_1 and $\mathbf{p}' \Vdash^{\text{call}} \mathbf{f} \ e'_1 \cdots e'_n \Rightarrow o'$ in T_2 ; we can use the first part of the induction hypothesis to conclude $o \wr o'$.
- $s = \mathbf{f} \ s_1 \cdots s_n$ with $n > \text{arity}_{\mathbf{p}}(\mathbf{f})$; then both conclusions follow by [Applied]. There are $e_1, \dots, e_n, e'_1, \dots, e'_n$ such that, by the induction hypothesis, each $e_i \wr e'_i$. Moreover, there are u, u' such that T_1 has a subtree with root $\mathbf{p}' \Vdash^{\text{call}} \mathbf{f} \ e_1 \cdots e_k \Rightarrow u$ and T_2 has a subtree with root $\mathbf{p}' \Vdash^{\text{call}} \mathbf{f} \ e'_1 \cdots e'_k \Rightarrow u'$, where $k = \text{arity}_{\mathbf{p}}(\mathbf{f})$; by the induction hypothesis, clearly $u \wr u'$, and since there are o_2, o'_2 such that $u(e_{k+1}, \dots, e_n) \ni o_2 \sqsupseteq o$ and $u'(e'_{k+1}, \dots, e'_n) \ni o'_2 \sqsupseteq o'$, the induction argument in the variable case provides $o_2 \wr o'_2$, so $o \wr o'$ by Lemma E19. \square

This result implies that all (non-deterministic) extensional values in the derivation tree are *internally consistent*: $o \wr o$.

Now, recall that our mission is to transform a derivation tree for $\mathbf{p}' \Vdash \text{start } d_1 \cdots d_M \Rightarrow b$ into one which uses only deterministic extensional values. A key step to this will be to define a deterministic extensional value $o' \sqsupseteq o$ for every non-deterministic extensional value o occurring in the tree. Using consistency, we can do that; o' is chosen to be $\sqcup\{o\}$ defined below:

Definition 25. *Given a non-empty, consistent set X —that is, $\emptyset \neq X \subseteq \llbracket \sigma \rrbracket_{\mathcal{B}}$ with $e \wr u$ for all $e, u \in X$ —let $\sqcup X \in \llbracket \sigma \rrbracket_{\mathcal{B}}$ be defined as follows:*

- if $\sigma \in \mathcal{S}$, then by consistency X can only have one element; we let $\sqcup\{d\} = d$;
- if $\sigma = \sigma_1 \times \sigma_2$, then $\sqcup X = (\sqcup\{e \mid (e, u) \in X\}, \sqcup\{u \mid (e, u) \in X\})$
(this is well-defined because $(e_1, u_1) \wr (e_2, u_2)$ implies both $e_1 \wr e_2$ and $u_1 \wr u_2$, so indeed the two sub-sets are consistent)
- if $\sigma = \sigma_1 \Rightarrow \tau$, then $\sqcup X = \{(e, \sqcup Y_e) \mid e \in \llbracket \sigma \rrbracket_{\mathcal{B}} \wedge Y_e = \bigcup_{A_\sigma \in X} \{o \mid (u, o) \in A \wedge e \sqsupseteq \sqcup\{u\}\} \wedge Y_e \neq \emptyset\}_{\sigma_1 \Rightarrow \sigma_2}$
(this is well-defined because for every e there is only one Y_e , and Y_e is indeed consistent: if $o_1, o_2 \in Y$, then there are $A_\sigma^{(1)}, A_\sigma^{(2)} \in X$ and there exist u_1, u_2 such that $(u_1, p_1) \in A^{(1)}, (u_2, o_2) \in A^{(2)}$ and both $e \sqsupseteq u_1$ and $e \sqsupseteq u_2$; by Lemma E19—using that $e \wr e$ because $e \in \llbracket \sigma \rrbracket_{\mathcal{B}}$ —the latter implies that $u_1 \wr u_2$, so by consistency of $A_\sigma^{(1)}$ and $A_\sigma^{(2)}$ indeed $o_1 \wr o_2$)

Now, the transformation of a **choose**-free—and therefore consistent—derivation tree for $\mathbf{p}' \Vdash^{\text{call}} \text{start } d_1 \cdots d_M \Rightarrow b$ into one which uses only deterministic extensional values is detailed in Lemma E24. The transformation is mostly done by a fairly straightforward induction on the depth of the tree (or more precisely, on the maximum depth of a set of trees), but to guarantee correctness we will be obliged to assert a number of properties of \sqcup . This is done in Lemmas E22–E23.

To start, we derive a kind of monotonicity for \sqcup with respect to \sqsupseteq :

Lemma E21. *Let $X, Y \subseteq \llbracket \sigma \rrbracket_{\mathcal{B}}$ be non-empty consistent sets, and suppose that for every $e \in Y$ there is some $e' \in X$ such that $e' \sqsupseteq e$. Then $\sqcup X \sqsupseteq \sqcup Y$.*

In particular, if $X \supseteq Y$ then $\sqcup X \sqsupseteq \sqcup Y$.

Proof. The second statement follows immediately from the first, since \sqsupseteq is reflexive. For the first statement, we use induction on the form of σ .

If $\sigma \in \mathcal{S}$ there is little to prove: X and Y contain the same single element.

If $\sigma = \sigma_1 \times \sigma_2$, then $\sqcup X = (\sqcup\{u \mid (u, o) \in X\}, \sqcup\{o \mid (u, o) \in X\})$ and $\sqcup Y = (\sqcup\{u \mid (u, o) \in Y\}, \sqcup\{o \mid (u, o) \in Y\})$. Since, for every u in $\{u \mid (u, o) \in Y\}$ there is some $(u', o') \in X$ with $u' \sqsupseteq u$ (by definition of \sqsupseteq for pairs), the containment property also holds for the first sub-set; it is as easily obtained for the second. Thus we complete by the induction hypothesis and the definition of \sqsupseteq .

Otherwise $\sigma = \sigma_1 \Rightarrow \sigma_2$; denote $\sqcup X = A_\sigma$ and $\sqcup Y = B_\sigma$. Now, all elements of B can be written as $(u, \sqcup Y_u)$ where $Y_u = \bigcup_{D_\sigma \in Y} \{o \mid (u', o) \in D \wedge u \sqsupseteq \sqcup\{u'\}\}$, and all elements of A as $(u, \sqcup X_u)$, where $X_u = \bigcup_{C_\sigma \in X} \{o \mid (u', o) \in C \wedge u \sqsupseteq \sqcup\{u'\}\}$. Let $(u, \sqcup Y_u) \in B$; we claim that (1) X_u is non-empty, (2) $(u, \sqcup X_u) \in A$ and (3) $\sqcup X_u \sqsupseteq \sqcup Y_u$, which suffices to conclude $\sqcup X \sqsupseteq \sqcup Y$.

1. $(u, \sqcup Y_u) \in B$ gives that Y_u is non-empty, so it has at least one element o with $(u', o) \in D$ for some $D_\sigma \in Y$; by assumption, there is $C_\sigma \in X$ with $C_\sigma \sqsupseteq D_\sigma$, which implies that $(u', o') \in C_\sigma$ for some $o' \sqsupseteq o$; as $u \sqsupseteq u'$ we have $o' \in X_u$;
2. follows from (1);
3. for all $o \in Y_u$, there are u' with $u \sqsupseteq u'$ and $D_\sigma \in Y$ such that $(u', o) \in D$, and by assumption $C_\sigma \in X$ and $(u', o') \in C$ with $o' \sqsupseteq o$; as $u \sqsupseteq u'$, we have $o' \in X_u$. The induction hypothesis therefore gives $\sqcup X_u \sqsupseteq \sqcup Y_u$. \square

We can think of \sqcup as defining a kind of *supremum*: $\sqcup X$ is the supremum of the set $\{\sqcup\{e\} \mid e \in X\}$ with regards to the ordering relation \sqsupseteq . The first part of this is given by Lemma E21: $\sqcup X$ is indeed greater than $\sqcup\{e\}$ for all $e \in X$ because $X \sqsupseteq \{e\}$. The second part, that $\sqcup X$ is the *smallest* deterministic extensional value with this property, holds by Lemma E22:

Lemma E22. *Let $X = X^{(1)} \cup \dots \cup X^{(n)} \subseteq \llbracket \sigma \rrbracket_{\mathcal{B}}$ be a consistent set with $n > 0$ and all $X^{(i)}$ non-empty, and let $e \in \langle \sigma \rangle_{\mathcal{B}}$ be such that $e \sqsupseteq \sqcup X^{(i)}$ for all $1 \leq i \leq n$. Then $e \sqsupseteq \sqcup X$.*

Proof. By induction on the form of σ .

If $\sigma \in \mathcal{S}$, then each $\sqcup X^{(i)} = e$; thus, $X^{(1)} = \dots = X^{(n)} = X = \{e\}$ and $\sqcup X = e$ as well.

If $\sigma = \sigma_1 \times \sigma_2$, then $e = (e_1, e_2)$ and $\sqcup X = (\sqcup Y_1, \sqcup Y_2)$, where $Y_j = \{u_j \mid (u_1, u_2) \in X\}$ for $j \in \{1, 2\}$. Let $Y_j^{(i)} = \{u_j \mid (u_1, u_2) \in X^{(i)}\}$. Then clearly each $Y_j = Y_j^{(1)} \cup \dots \cup Y_j^{(n)}$, and $e \sqsupseteq \sqcup X^{(i)}$ implies that each $e_j \sqsupseteq \sqcup Y_j^{(i)}$. The induction hypothesis gives $e_j \sqsupseteq \sqcup Y_j$ for both j .

If $\sigma = \sigma_1 \Rightarrow \sigma_2$, then write $e = A_\sigma$. Now,

- for $u \in \langle \sigma_1 \rangle_{\mathcal{B}}$, denote $Y_u^{(i)} = \bigcup_{B_\sigma \in X^{(i)}} \{o \mid (u', o) \in B \wedge u \sqsupseteq \sqcup\{u'\}\}$;
- for $(u, \sqcup Y_u) \in \sqcup X$, we can write $Y = Y_u^{(1)} \cup \dots \cup Y_u^{(N)}$;
- for $(u, \sqcup Y_u) \in \sqcup X$, some $Y_u^{(i)}$ must be non-empty;
- as $(u, \sqcup Y_u^{(i)}) \in \sqcup X^{(i)}$, there exists $(u, o') \in A$ with $o' \sqsupseteq \sqcup Y_u^{(i)}$;
- as there is only one o' with $(u, o') \in A$, we obtain $o' \sqsupseteq \sqcup Y_u^{(j)}$ for all non-empty $Y_u^{(j)}$;
- by the induction hypothesis, $o' \sqsupseteq \sqcup(Y_u^{(1)} \cup \dots \cup Y_u^{(N)}) = \sqcup Y_u$.

Thus, $A_\sigma \sqsupseteq \sqcup X$ as required. \square

The next helper result concerns application of deterministic extensional values as (partial) functions. Very roughly, we see that if e is at least the “supremum” of $\{e^{(1)}, \dots, e^{(m)}\}$, then the result $c \in e(u_1, \dots, u_n)$ of applying e to some extensional values u_1, \dots, u_n is at least as large as each $\sqcup e^{(j)}(u_1, \dots, u_n)$.

The lemma is a bit broader than this initial sketch, however, as it also allows for the $e^{(j)}$ to be applied on smaller $u_i^{(j)}$; i.e., we actually show that $c \sqsupseteq \sqcup e^{(j)}(u_1^{(j)}, \dots, u_n^{(j)})$ if each $u_i \sqsupseteq \sqcup\{u_i^{(j)} \mid 1 \leq j \leq m\}$. Formally:

Lemma E23. *Let $n \geq 0$ and suppose that:*

- $\langle \sigma_1 \Rightarrow \dots \Rightarrow \sigma_n \Rightarrow \tau \rangle_{\mathcal{B}} \ni e \sqsubseteq \sqcup \{e^{(1)}, \dots, e^{(m)}\};$
- $\langle \sigma_i \rangle_{\mathcal{B}} \ni u_i = \sqcup \{u_i^{(1)}, \dots, u_i^{(m)}\}$ for $1 \leq i \leq n;$
- $e^{(j)}(u_1^{(j)}, \dots, u_n^{(j)}) \ni c^{(j)} \sqsubseteq o^{(j)}$ for $1 \leq j \leq m;$
- $o = \sqcup \{o^{(1)}, \dots, o^{(m)}\}.$

Then there exists $c \in \langle \tau \rangle_{\mathcal{B}}$ such that $e(u_1, \dots, u_n) \ni c \sqsubseteq o$.

Proof. By induction on n . First suppose that $n = 0$, so each $e^{(j)} = c^{(j)} \sqsubseteq o^{(j)}$. Then $e \sqsubseteq \sqcup \{e^{(1)}, \dots, e^{(m)}\} \sqsubseteq \sqcup \{o^{(1)}, \dots, o^{(m)}\} = o$ by Lemma E21, so $e() \ni e \sqsubseteq o$ by transitivity of \sqsubseteq .

Now let $n > 0$. For $1 \leq j \leq m$ the third observation gives $A^{(j)}$ such that $e^{(j)}(u_1^{(j)}, \dots, u_n^{(j)}) \ni A_{\sigma_n \Rightarrow \tau}^{(j)}$ and $(u_n^{(j)}, c^{(j)}) \in A^{(j)}$. Then by the induction hypothesis, there exists $A_{\sigma_n \Rightarrow \tau} \in e(u_1, \dots, u_{n-1})$ such that $A_{\sigma_n \Rightarrow \tau} \sqsubseteq \sqcup \{A^{(1)}, \dots, A^{(m)}\}$. That is, omitting the subscript n :

- $\langle \sigma \Rightarrow \tau \rangle_{\mathcal{B}} \ni A_{\sigma \Rightarrow \tau} \sqsubseteq \sqcup \{A_{\sigma \Rightarrow \tau}^{(1)}, \dots, A_{\sigma \Rightarrow \tau}^{(m)}\};$
- $\langle \sigma \rangle_{\mathcal{B}} \ni u = \sqcup \{u^{(1)}, \dots, u^{(m)}\};$
- for $1 \leq j \leq m$: $(u^{(j)}, c^{(j)}) \in A^{(j)}$ for some $c^{(j)} \sqsubseteq o^{(j)};$
- $o = \sqcup \{o^{(1)}, \dots, o^{(m)}\}.$

Moreover, for every c such that $(u, c) \in A$ also $c \in e(u_1, \dots, u_n)$; thus, we are done if we can identify such $c \sqsubseteq o$.

Let $B_u^{(j)} := \{o' \mid (u', o') \in A^{(j)} \wedge u \sqsubseteq \sqcup \{u'\}\}$ and let $B_u := B_u^{(1)} \cup \dots \cup B_u^{(m)}$. Then we have:

- $c^{(j)} \in B_u$ for $1 \leq j \leq m$: since $(u^{(j)}, c^{(j)}) \in A^{(j)}$, and $u = \sqcup \{u^{(1)}, \dots, u^{(m)}\} \sqsubseteq \sqcup \{u^{(j)}\}$ by Lemma E21, we have $c^{(j)} \in B_u^{(j)} \subseteq B_u;$
- since therefore $B_u \neq \emptyset$, the pair $(u, \sqcup B_u)$ occurs in the set underlying $\sqcup \{A^{(1)}, \dots, A^{(m)}\};$
- since $A_{\sigma \Rightarrow \tau} \sqsubseteq \sqcup \{A_{\sigma \Rightarrow \tau}^{(1)}, \dots, A_{\sigma \Rightarrow \tau}^{(m)}\}$, there exists $c \sqsubseteq \sqcup B_u$ such that $(u, c) \in A;$
- since $A_{\sigma \Rightarrow \tau} \in \langle \sigma_n \Rightarrow \tau \rangle_{\mathcal{B}}$, there is only one choice for $c;$
- $c \sqsubseteq \sqcup B_u \sqsubseteq \sqcup \{c^{(j)}\} \sqsubseteq \sqcup \{o^{(j)}\}$ for all $1 \leq j \leq m$ by Lemmas E21 and E21;
- therefore $c \sqsubseteq o$ by Lemma E22. \square

At last, all preparations done. We now turn to the promised proof that in a deterministic setting, it suffices to consider deterministic extensional values.

Lemma E24. *If \mathbf{p} is deterministic and $\mathbf{p}' \vdash^{\text{call}} \text{start } d_1 \dots d_M \Rightarrow b$, then this can be derived using a functional tree: a derivation tree where all extensional values are in some $\langle \sigma \rangle_{\mathcal{B}}$.*

Proof. Let \mathbf{p}' be deterministic (so also \mathbf{p}' is). For the sake of a stronger induction hypothesis, it turns out to be useful to use induction on *sets* of derivation trees, rather than a single tree. Specifically, we prove the following statements:

1. Suppose T_1, \dots, T_N are derivation trees, and there are fixed \mathbf{f}, n such that each tree T_j has a root $\mathbf{p}' \Vdash \mathbf{f} \ e_1^{(j)} \dots e_n^{(j)} \Rightarrow o^{(j)}$, where $e_i^{(j)} \wr e_i^{(k)}$ for all $1 \leq j, k \leq N$ and $1 \leq i \leq n$. Let e_1, \dots, e_n be deterministic extensional values such that $e_i \sqsupseteq \sqcup \{e_i^{(j)} \mid 1 \leq j \leq N\}$ for $1 \leq i \leq n$. We can derive $\mathbf{p}' \Vdash^{\text{call}} \mathbf{f} \ e_1 \dots e_n \Rightarrow o := \sqcup \{o^{(j)} \mid 1 \leq j \leq N\}$ by a functional tree.
2. Suppose T_1, \dots, T_N are derivation trees, and there is some fixed s such that each tree T_j has a root $\mathbf{p}', \eta^{(j)} \Vdash s \Rightarrow o^{(j)}$, where $\eta^{(j)}(x) \wr \eta^{(k)}(x)$ for all $1 \leq j, k \leq N$ and variables x in the shared domain. Let η be an ext-environment on the same domain mapping to functional extensional values such that $\eta(x) \sqsupseteq \sqcup \{\eta^{(j)}(x) \mid 1 \leq j \leq N\}$ for all x . Writing $o := \sqcup \{o^{(j)} \mid 1 \leq j \leq N\}$, we can derive $\mathbf{p}', \eta \Vdash s \Rightarrow o$ by a functional tree. (We assume that no sub-expression of s has an if-then-else at the head of an application.)

The first of these claims proves the lemma for $N = 1$: clearly data expressions are self-consistent, and the only $o \sqsupseteq b = \sqcup \{b\}$ is b itself, so the claim says that the root can be derived using a functional tree.

We prove the claims together by a shared induction on the maximum depth of any T_j . We start with the first claim. There are two cases:

- $n = \text{arity}_{\mathbf{p}}(\mathbf{f})$: then for each T_j there is a clause $\rho_j: \mathbf{f} \ \ell_1 \dots \ell_n = s$ which imposes $\eta^{(j)}$ such that the immediate subtree of T_j is $\mathbf{p}', \eta^{(j)} \Vdash o^{(j)}$.
Now, let $\ell: \sigma$ be a linear pattern, η an ext-environment and $e, u \in \llbracket \sigma \rrbracket_{\mathcal{B}}$ be such that $e \wr u$ and $\ell\eta = e$. By a simple induction on the form of ℓ we find an ext-environment η' on domain $\text{Var}(\ell)$ such that $\ell\eta' = u$ and $\eta(x) \wr \eta'(x)$.
Thus, the first matching clause ρ_j is necessarily the same for all T_j , and we have $\eta^{(j)}(x) \wr \eta^{(k)}(x)$ for all j, k, x . For all $1 \leq i \leq n$ and $1 \leq j \leq N$, we have $e_i^{(j)} = \ell_i \eta^{(j)}$. Another simple induction on ℓ_i proves that we can find η with each $\eta(x) \sqsupseteq \sqcup \{\eta^{(j)}(x) \mid 1 \leq j \leq N\}$ such that $e_i = \ell_i \eta$.
The induction hypothesis gives $\mathbf{p}', \eta \Vdash o$, so $\mathbf{f} \ e_1 \dots e_n \Rightarrow o$ by [Call].
- $n < \text{arity}_{\mathbf{p}}(\mathbf{f})$: each of the trees T_j is derived by [Value]. Write $o = O_\sigma$ and $o^{(j)} = O_\sigma^{(j)}$ for $1 \leq j \leq N$. We are done by [Value] if $\mathbf{p}' \Vdash^{\text{call}} \mathbf{f} \ e_1 \dots e_n \ e_{n+1} \Rightarrow o'$ for all $(e_{n+1}, o') \in O$.
Since $o = \sqcup \{o^{(1)}, \dots, o^{(N)}\}$, we can write $o' = \sqcup Y_{e_{n+1}}$ and identify a non-empty set $\text{Pairs}_{e_{n+1}} = \{(e, u) \in O^{(1)} \cup \dots \cup O^{(N)} \mid e_{n+1} \sqsupseteq \sqcup \{e\}\}$ such that $Y_{e_{n+1}} = \{u \mid (e, u) \in \text{Pairs}_{e_{n+1}}\}$.
For each element (e, u) of Pairs_C , some T_j has a subtree with root $\mathbf{p}' \Vdash^{\text{call}} \mathbf{f} \ e_1^{(j)} \dots e_n^{(j)} \ e \Rightarrow u$. Let $\text{Trees}_{e_{n+1}}$ be the corresponding set of trees, and note that all trees in $\text{Trees}_{e_{n+1}}$ have a strictly smaller depth than the T_j they originate from, so certainly smaller than the maximum depth.
Now, for $1 \leq i \leq n+1$, let $\text{Args}_i := \{\text{argument } i \text{ of the root of } T \mid T \in \text{Trees}_{e_{n+1}}\}$. We observe that:
 - for $1 \leq i \leq n$: $e_i \sqsupseteq \sqcup \text{Args}_i$: we have $\text{Args}_i \subseteq \{e_i^{(1)}, \dots, e_i^{(N)}\}$, so by Lemma E21, $e_i \sqsupseteq \sqcup \{e_i^{(j)} \mid 1 \leq j \leq N\} \sqsupseteq \sqcup \text{Args}_i$, which suffices by transitivity (Lemma E13);
 - $e_{n+1} \sqsupseteq \sqcup \text{Args}_{j+1}$: $e_{n+1} \sqsupseteq \{e\}$ for all $e \in \text{Args}_{j+1}$, so this is given by Lemma E22.

- $o' = \sqcup Y_{e_{n+1}} = \sqcup \{\text{right-hand sides of the roots of } \text{Trees}_{e_{n+1}}\}$.
Therefore $\mathbf{p}' \Vdash^{\text{call}} \mathbf{f} \ e_1 \cdots e_n \ e_{n+1} \Rightarrow o'$ by the induction hypothesis as required.

For the second case, consider the form of s .

- $s = \mathbf{c} \ s_1 \cdots s_m$ with $\mathbf{c} \in \mathcal{C}$: then each $o^{(i)} = s\eta^{(i)} \in \mathcal{B}$, so $o = o^{(1)} = \cdots = o^{(N)}$ and—since the variables in s all have order 0—we have $\eta(x) = \eta^{(1)}(x) = \cdots = \eta^{(N)}(x)$ for all relevant x . Thus also $o = s\eta$ and we complete by [Constructor].
- $s = (s_1, s_2)$; each tree T_j has two immediate subtrees: one with root $\mathbf{p}', \eta^{(j)} \Vdash s_1 \Rightarrow o_1^{(j)}$ and one with root $\mathbf{p}', \eta^{(j)} \Vdash s_2 \Rightarrow o_2^{(j)}$, where $o^{(j)} = (o_1^{(j)}, o_2^{(j)})$.
We can write $o = (o_1, o_2)$ where $o_1 = \sqcup \{o_1^{(j)} \mid 1 \leq j \leq N\}$ and $o_2 = \sqcup \{o_2^{(j)} \mid 1 \leq j \leq N\}$, and as the induction hypothesis for both subtrees gives $\mathbf{p}', \eta \Vdash s_1 \Rightarrow o_1$ and $\mathbf{p}', \eta \Vdash s_2 \Rightarrow o_2$ respectively, we conclude $\mathbf{p}', \eta \Vdash s \Rightarrow o$ by [Pair].
- $s = \text{if } s_1 \text{ then } s_2 \text{ else } s_3$: for each tree T_j , the first subtree has the form $\mathbf{p}', \eta^{(j)} \Vdash s_1 \Rightarrow \text{true}$ or $\mathbf{p}', \eta^{(j)} \Vdash s_1 \Rightarrow \text{false}$; by consistency of derivation trees (Lemma E20), either **true** or **false** is chosen for *all* these subtrees. We assume the former; the latter case is symmetric.
By the induction hypothesis for this first subtree, $\mathbf{p}', \eta \Vdash s_1 \Rightarrow \text{true} = \sqcup \{\text{true}, \dots, \text{true}\}$ as well.
The second immediate subtree of all trees T_j has a root of the form $\mathbf{p}', \eta^{(j)} \Vdash s_2 \Rightarrow o^{(j)}$. By the induction hypothesis for this second subtree, $\mathbf{p}', \eta \Vdash s_2 \Rightarrow o$. Thus we conclude $\mathbf{p}', \eta \Vdash s \Rightarrow o$ by [Cond-True].
- $s = x \ s_1 \cdots s_n$ with $x \in \mathcal{V}$: each of the trees T_j has n subtrees of the form $\mathbf{p}', \eta^{(j)} \Vdash s_1 \Rightarrow e_i^{(j)}$ (for $1 \leq i \leq n$); by the induction hypothesis, we have $\mathbf{p}', \eta \Vdash s_i \Rightarrow e_i$, where $e_i = \sqcup \{e_i^{(j)} \mid 1 \leq j \leq N\}$. But then:
 - $\langle \sigma_1 \Rightarrow \cdots \Rightarrow \sigma_n \Rightarrow \tau \rangle_{\mathcal{B}} \ni \eta(x) \sqsupseteq \sqcup \{\eta^{(1)}(x), \dots, \eta^{(N)}(x)\}$;
 - $\langle \sigma_i \rangle_{\mathcal{B}} \ni e_i = \sqcup \{e_i^{(1)}, \dots, e_i^{(N)}\}$ for $1 \leq i \leq n$;
 - there are $u^{(j)}$ such that $\eta^{(j)}(e_1^{(j)}, \dots, e_n^{(j)}) \ni u^{(j)} \sqsupseteq o^{(j)}$ for $1 \leq j \leq N$;
 - $o = \sqcup \{o^{(1)}, \dots, o^{(N)}\}$.
 By Lemma E23, there exists $u \in o(e_1, \dots, e_n)$ such that $u \sqsupseteq o$. We conclude $\mathbf{p}', \eta \Vdash s \Rightarrow o$ by [Variable].
- $s = \mathbf{f} \ s_1 \cdots s_n$ with $n \leq \text{arity}_{\mathbf{p}}(\mathbf{f})$: then necessarily each $\mathbf{p}', \eta^{(j)} \Vdash s \Rightarrow o^{(j)}$ follows by [Func]. Thus, for $1 \leq j \leq N$ there are $e_1^{(j)}, \dots, e_n^{(j)}$ such that:
 - $\mathbf{p}', \eta^{(j)} \Vdash s_i \Rightarrow e_i^{(j)}$ for $1 \leq i \leq n$ and
 - $\mathbf{p}' \Vdash^{\text{call}} \mathbf{f} \ e_1^{(j)} \cdots e_n^{(j)} \Rightarrow o^{(j)}$.
 Now, clearly each set $\{e_i^{(j)} \mid 1 \leq j \leq N\}$ is consistent by the simple fact that there are derivation trees for them: this is the result of Lemma E20. Defining $e_i := \sqcup \{e_i^{(1)}, \dots, e_i^{(N)}\}$ for $1 \leq i \leq n$, the induction hypothesis gives that $\mathbf{p}', \eta \Vdash s_i \Rightarrow e_i$, and that $\mathbf{p}' \Vdash^{\text{call}} \mathbf{f} \ e_1 \cdots e_n \Rightarrow o$, all by functional trees. We complete with [Func].
- $s = \mathbf{f} \ s_1 \cdots s_n$ with $n > k := \text{arity}_{\mathbf{p}}(\mathbf{f})$: then there are $e_1^{(j)}, \dots, e_n^{(j)}, u^{(j)}, c^{(j)}$ such that for all $1 \leq j \leq N$:

- tree T_j has subtrees $\mathbf{p}', \eta^{(j)} \Vdash s_i \Rightarrow e_i^{(j)}$ for $1 \leq i \leq n$;
- tree T_j has a subtree $\Vdash^{\text{call}} \mathbf{f} e_1^{(j)} \dots e_k^{(j)} \Rightarrow u^{(j)}$;
- $u^{(j)}(e_{k+1}^{(j)}, \dots, e_n^{(j)}) \ni c^{(j)} \sqsubseteq o^{(j)}$.

Therefore, by the induction hypothesis and Lemma E23, we can identify e_1, \dots, e_n, u, c such that:

- $e_i = \sqcup \{e_i^{(1)}, \dots, e_i^{(N)}\}$ and $\mathbf{p}', \eta \Vdash s_i \Rightarrow e_i$ for $1 \leq i \leq n$;
- $\mathbf{p}' \Vdash^{\text{call}} \mathbf{f} e_1 \dots e_k \Rightarrow u = \sqcup \{u^{(1)}, \dots, u^{(N)}\}$;
- $u(e_{k+1}, \dots, e_n) \ni c \sqsubseteq o$.

Therefore $\mathbf{p}', \eta \Vdash s \Rightarrow o$ by [Apply]. \square

With this, the one remaining lemma—completeness of Algorithm 7—is trivial.

Lemma 19. *If $\llbracket \mathbf{p} \rrbracket(d_1, \dots, d_M) \mapsto b$ and \mathbf{p} is deterministic, then Algorithm 7 returns a set $A \cup \{b\}$.*

Proof. Suppose $\llbracket \mathbf{p} \rrbracket(d_1, \dots, d_M) \mapsto b$ for a deterministic program \mathbf{p} . By Lemma E17, we can derive $\Vdash^{\text{call}} \text{start } d_1 \dots d_M \Rightarrow b$. By Lemma E24, this can be derived by a tree which only uses deterministic extensional values. By Lemma E18, Algorithm 7 therefore returns a set containing b . \square

F Arrow depth and unitary variables (Section 7)

In Sections 5 and 6, we have demonstrated two things:

- that cons-free deterministic programs of data order K characterise $\text{EXP}^K \text{TIME}$
- that cons-free non-deterministic programs of data order $K > 0$ characterise ELEMENTARY

However, most of the proof effort has gone towards the complexity and correctness of the simulation algorithm—arguably the least interesting side, since the characterisation result for deterministic programs is a natural extension of an existing result of [12], while the surprising result for non-deterministic programs is that we get *at least* ELEMENTARY, not that we cannot go beyond.

The efforts pay off, however, when we consider what is needed to recover the original hierarchy. The proofs require very little adaptation to obtain Theorems 3 and 4. We start with Theorem 3, which we split up in its two parts.

Lemma F25. *Every decision problem in $\text{EXP}^K \text{TIME}$ is accepted by a deterministic cons-free program with data arrow depth K .*

Here, a program has *data arrow depth* K if all variables are typed with a type of arrow depth K .

Proof. Both Lemma 4 and 5 also apply if “data order K ” is replaced by “data arrow depth K ”. With this observation, we may copy the proof of Lemma 6. \square

Lemma F26. *Every decision problem accepted by a deterministic cons-free program \mathbf{p} with data arrow depth K is in $\text{EXP}^K \text{TIME}$.*

Proof. Defining a type to be “proper” if its arrow depth is smaller than K , types of order 0 are proper and $\sigma \times \tau$ is proper iff both σ and τ are. Thereofre, all the proofs in Appendix A extend to arrow depth, and we immediately obtain a variation of Lemma 1 where data order is replaced by data arrow depth. Note that Lemma 1 considers the transformation from \mathbf{p} to \mathbf{p}' in both algorithms.

Now, in Algorithm 13, alter step 1a by using the transformation which considers arrow depth rather than data order, and in step 1b, only include statements $\mathbf{f} \ e_1 \cdots e_n \leadsto o$ if $\text{depth}(\sigma_{n+1} \Rightarrow \dots \Rightarrow \sigma_m \Rightarrow \kappa') \leq K$ (rather than considering $\text{ord}()$). This does not affect correctness of the algorithm, as is easily checked by going over the proofs of Appendix E: the only place in the algorithm where it may be important whether any statements $\mathbf{f} \ e_1 \cdots e_n \leadsto o$ were removed is step 2g, but here only calls with an output arrow depth $\leq K$ may be used (due to the preparation step 1a and the altered Lemma 1).

Moreover, by the combination of Lemma 8 (which also applies to the thus modified algorithm) and Lemma 14, this altered algorithm finds the possible results of \mathbf{p} on given input in $\text{TIME}(a \cdot \exp_2^K(n^b))$ for some a, b . Therefore, any decision problem accepted by \mathbf{p} is in $\text{EXP}^K \text{TIME}$. \square

We thus conclude:

Theorem 3. *The class of non-deterministic cons-free programs where all variables are typed with a type of arrow depth K characterises $\text{EXP}^K \text{TIME}$.*

Proof. By the combination of Lemmas F25 and F26. \square

We turn to Theorem 4, which considers programs with unitary variables. Again, one direction—the minimum power of such programs—is quite simple:

Lemma F27. *Every decision problem in $\text{EXP}^K \text{TIME}$ is accepted by a deterministic cons-free program with data arrow depth K and unitary variables.*

Proof. All variables employed in both Lemma 4 and Lemma 5 have a type that is either a sort, or has the form $\sigma \Rightarrow \text{bool}$; thus, the simulation program is unitary, and we may copy the proof of Lemma 6. \square

The second part of Theorem 4 can once more be derived using a variation of Algorithm 13. However, here we must be a little careful: where both data order and arrow depth are *recursive* properties, the property that a type is “unitary” (i.e., of the form κ or $\sigma \Rightarrow \kappa$ with $\text{ord}(\kappa) = 0$) is not recursive. Thus, a unitary type of a fixed data order may still have an arbitrarily high arrow depth. We circumvent this problem by altering unused subtypes.

Lemma F28. *Every decision problem accepted by a deterministic cons-free program \mathbf{p} with data order K and unitary variables is in $\text{EXP}^K \text{TIME}$.*

Proof. Let a type σ be *proper* if $\text{ord}(\sigma) \leq K$ and (a) $\text{ord}(\sigma) = 0$ or (b) σ has the form $\tau \Rightarrow \kappa$ with $\text{ord}(\kappa) = 0$ or (c) σ has the form $\sigma_1 \times \sigma_2$ with both σ_1 and σ_2 proper. This notion of properness has the properties described in Definition 20, so the proofs in Appendix A extend; \mathbf{p} can be transformed into a program \mathbf{p}' with data order K and unitary variables such that for all clauses $\mathbf{f} \ \ell_1 \cdots \ell_k = s$: all sub-expressions t of s have a unitary type with order $\leq K$.

Now let $fixtype$ be defined as follows:

- $fixtype(\iota) = \iota$ for $\iota \in \mathcal{S}$
- $fixtype(\sigma \times \tau) = fixtype(\sigma) \times fixtype(\tau)$
- $fixtype(\sigma_1 \Rightarrow \dots \Rightarrow \sigma_n \Rightarrow \kappa) = fixtype(\sigma_1) \Rightarrow \kappa$ if $ord(\kappa) = 0$ and $n > 0$.

Then clearly $depth(fixtype(\sigma)) \leq K$ whenever $ord(\sigma) \leq K$. Given type assignments \mathcal{F} (for defined symbols and data constructors) and Γ for variables, let $\mathcal{F}' := \{\mathbf{f} : fixtype(\sigma_1) \Rightarrow \dots \Rightarrow fixtype(\sigma_m) \Rightarrow fixtype(\kappa) \mid \mathbf{f} : \sigma_1 \Rightarrow \dots \Rightarrow \sigma_m \Rightarrow \kappa \in \mathcal{F}\}$ and $\Gamma' := \{x : fixtype(\sigma_1) \Rightarrow \dots \Rightarrow fixtype(\sigma_m) \Rightarrow fixtype(\kappa) \mid x : \sigma_1 \Rightarrow \dots \Rightarrow \sigma_m \Rightarrow \kappa \in \Gamma\}$. Now suppose that all clauses in \mathbf{p}' are well-typed under \mathcal{F}' and the corresponding type environment Γ' . Since typing does not affect the semantics of Figure 6, $\llbracket \mathbf{p} \rrbracket(d_1, \dots, d_M) \mapsto b$ if and only if $\llbracket \mathbf{p}' \rrbracket(d_1, \dots, d_M) \mapsto b$ still holds. Using Lemma F26—and the observation that the translation from \mathbf{p} to \mathbf{p}' takes constant time as it does not consider the input d_1, \dots, d_M —any decision problem accepted by \mathbf{p} is therefore in EXP^KTIME .

It remains to be seen that every clause $\mathbf{f} \ell_1 \dots \ell_k = s$ which is well-typed under \mathcal{F} with type environment Γ is also well-typed using \mathcal{F}' and Γ' instead. To see this, we prove the following by induction on the size of s :

Suppose variables have a proper type, and let $s : \sigma$ using \mathcal{F}, Γ . If for all $t \trianglelefteq s$, the type of t is proper w.r.t. \mathcal{F}, Γ and t does not have the form $(\text{if } b \text{ then } s_1 \text{ else } s_3) t_1 \dots t_n$ or $(\text{choose } s_1 \dots s_m) t_1 \dots t_n$ with $n > 0$, then $s : fixtype(\sigma)$ using \mathcal{F}', Γ' .

- If $s = \text{choose } s_1 \dots s_m$, then each $s_i : \sigma$ using \mathcal{F}, Γ , so by the induction hypothesis each $s_i : fixtype(\sigma)$ using \mathcal{F}', Γ' ; this gives $s : fixtype(\sigma)$ following the typing rules for **choose**.
- If $s = \text{if } b \text{ then } s_1 \text{ else } s_2$, then by the induction hypothesis (and using that $fixtype(\text{bool}) = \text{bool}$), $b : \text{bool}$ and both $s_1 : fixtype(\sigma)$ and $s_2 : fixtype(\sigma)$.
- If $s = \mathbf{c} s_1 \dots s_m$, then $\sigma \in \mathcal{S}$ and we can write $\mathbf{c} : \kappa_1 \Rightarrow \dots \Rightarrow \kappa_m \Rightarrow \sigma \in \mathcal{F} \cap \mathcal{F}'$ where each κ_i has type order 0. By the induction hypothesis, each $s_i : fixtype(\kappa_i) = \kappa_i$ using \mathcal{F}', Γ' .
- If $s = a s_1 \dots s_n$ with $a \in \mathcal{V} \cup \mathcal{D}$, then a is typed with $\tau_1 \Rightarrow \dots \Rightarrow \tau_n \Rightarrow \sigma$ in $\mathcal{F} \cup \Gamma$. Since s has a proper type, we know that σ has the form κ or $\pi \Rightarrow \kappa$ or $\pi_1 \times \pi_2$; therefore $a : fixtype(\tau_1) \Rightarrow \dots \Rightarrow fixtype(\tau_n) \Rightarrow fixtype(\sigma) \in \mathcal{F}' \cup \Gamma'$. Since each $s_i : fixtype(\tau_i)$ by the induction hypothesis, we obtain $s : fixtype(\sigma)$.

Applying the result also to the $\mathbf{f} \ell_1 \dots \ell_k$, the entire clause is well-typed. \square